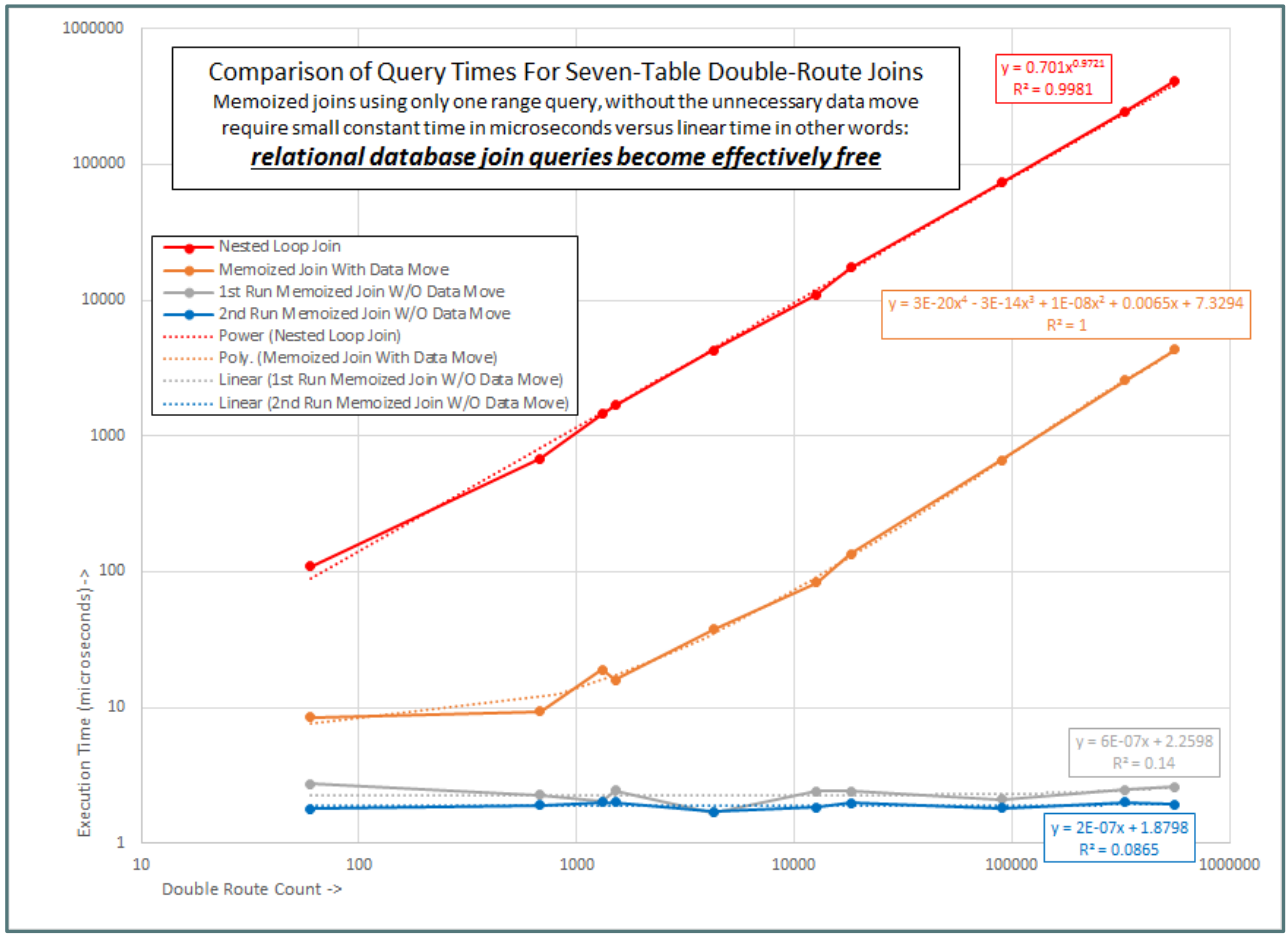
Kaldanes: A Different Method for Building Relational Database Systems That Scale Up and Out Massively

(Also, the fastest relational database system on Von Neumann hardware that you will ever see!) Version 1.0

C. Johnson, email: cjohnson@member.fsf.org



Not Burying the Lede (numbers from a HP small format i5-3570 box, running Centos-7, C++11, with no special hardware)

Table of Contents

1. Introduction

a. How to make a street-legal dragster

1) Start with a car or truck that you like and can be built upon.
2) Rip out everything that is excess weight: back seats, upholstery, spare tire, bumpers, hood, etc.
3) Replace the engine, drive train, chassis, wheels and tires with something that can do the job of accelerating to top speed while barely staying on the dragstrip.
4) Now very carefully put in just enough stuff to pass an inspection at the DMV with your very own street-legal dragster. [Also, put in lightweight air-conditioning (which you can turn off in racing) and a stereo system.]

b. Quick Technical Synopsis: How to make the fastest, most scalable relational database system using the C++ compiler and runtime as the data manager, and C++ as the DDL and DML database languages.

This technical white paper discusses the contents of the Kaldanes GitHub code base: https://github.com/charlesone/Kaldanes. That code is research code, and follows only one coding and design rule: performance and scalability of code at the hardware limit single-threaded, otherwise there is no point in making the code run parallel. Hence it will require refactoring: the purpose is to show what can be done, if we do things differently. "Differently" imposes new requirements upon what refactoring should be done.

It all starts from the ground up, building types, containers and methods that both perform and scale at the hardware limit:

- Slab allocation and deallocation of base + offset addressed database objects as automatic variables on the stack, or allocated using mmap:
    - This is much faster than fine-grained allocation, because moving the stack pointer up and down gigabytes or larger is speedy (apparently, g++ is using the kernel slab allocator for large objects, thank you!) and mmap uses kernel slab allocation and is also faster if you `MAP_POPULATE` and in future, `MAP_STACK` may avoid copying in assembling large databases. (The current code base does not use mmap.)

    - mmap makes the database objects shareable across processes within the node, or if the object is mmapped and fsynced to a file system, or CLFlushed to a Gen-Z memory-centric store (as it is in Linux For The Machine: L4TM), then it is also shareable across x86_64 or ARM operating systems like Linux, Windows or AOS (assuming compliance on fundamental type representation, byte order and small-endian bit order in C++, which is likely the case.)
- Making new template string classes that are base + offset and using the contiguous native arrays as containers, with anchors supported in the class objects for addressing using virtual addresses that are different (non-aligned) for every program using them and thus never stored in the objects:
    - **Direct** string class: an array of string elements of the same size, where the body of the string moves when sorted.
    - **Symbiont** string class: an array of string elements of the same size, where PMNK (poor man's normalized key) elements at the front or head of the string move when sorted, but the body of the string stays immutable and immovable in the slab.
    - **Head** string class: an array of string PMNK elements, which allocates a **Direct** string class array to hold the immutable and immovable string bodies.
- Performance comparison of these new string classes using various sorts, alongside std::string, revealed N-squared behavior for standard strings in interesting cases that might explain why C++ appears dramatically slower than C in applications like graph systems.

[Think of these as slab objects, or as they could be combined into single database slabs of objects, as enormous log records, or similar to b-tree blocks, that are directly addressable using virtual pointers constructed on-the-fly.]

Then we can construct a relational database system from variations on the fastest string classes:

- Using true static polymorphism, by hook or by crook, to avoid any use of the C++ V-Table, or the C++ keyword `virtual` in any inner loop.
- Using a combination of C++ static data, profligate use of `const`, `constexpr`, and `inline` when `constexpr` cannot be used, the compiler `-Ofast` option, variadic templates with multiple layers of parameter packs in recursion (SFINAE terminated) at compile time, to banish from runtime execution most of the relational database system's offline utilities such as query optimization and checking and metadata management, and also some of the related online functions … effectively forcing the relational deductive database into the compiler for:
    - Catalog, types, schema: what we call DDL.

    - Join query plan building and optimization, and a lot of the join execution: what we call DML.
- Conversion of the **Direct** string class into a **RowString** class with columns identified by `enum class Column`, and into arrays identified by `enum class Table`.
- Conversion of the **Head** string class into an **IndexString** friend class on those **RowString** columns.
- Creation of a **RelationVector** class to hold a directed pair of columns (from and to) connecting tables in join operations (only equijoins for the current demo programs.)
- Creation of a symbiotic pair of classes, a **JoinedRow** class to handle the arrayed output of relational joins and a **QueryPlan** class to hold and process the metadata, to do the query plan building and consistency/linkage checking to soft-define (on-the-fly) **JoinedRow** arrays from variadic template parameter packs of **RelationVector** objects containing pointers to the index objects, which produce pointers to the table rows, using nested loop joins.
- Since the **JoinedRow** is a very compact representation with one C++ `int` per column's table row, it becomes possible to, e.g., create ten-table join output objects of 40 bytes per ten joined table rows of output. Then a memoized join can be created by a full equijoin of all the rows for the parameter pack of **RelationVectors** across the entire database, providing for single range query sequential access by pointer to any join on the database. This delivers the functionality of cached materialized views at a tiny fraction of the space complexity (i.e., size.)
- Memoized joins could be indexed as well (not needed for the TableDemo, JoinDemo or BigJoinDemo applications in the code base) providing sequential "select" access to a joined row query ordered by any column of any table in the join, or combination keys for database selects. Since the database is in memory and there is no marshaling, buffering, messaging or I/O, the performance optimization provided by the database "project" operation is unnecessary, except on display operations.

Once again, following the dragster model, the current code base is research code: breaking, when necessary, any coding or design rule to further the goal of raw speed and scale. However, the goal is a street-legal dragster, and C++ template class interfaces can hide almost all of the C language ugliness for better composability of programs. However, the ugly necessities of static global variables, which may become unnecessary with advances in the C++ compiler (C++14, 17, 20, 23 …) that complex coding cannot be hidden for now. Also, the weirdness of parameter pack recursion in variadic templates needed for compile time generic programming by the mighty inference engine of C++11, which are probably a permanent feature of fast and scalable C++ code: that will require adaptation for survival by coders.

So, to repeat this a third time: the current code base is research code and refactoring will be necessitated, but must not be allowed to slow down the code or make it less scalable.

2. Why are speed and scale so critical?

The computing infrastructure of the world is now consuming power and resources at the scale of a large nation and that will grow, even as conservation reduces the resource consumption of actual human beings populating the world, if nothing is done to change the way we do business. As Pinar Tozun reminds us with her HPTS 2017 "hobby talk" [slides], we will get no more help from chip or power technology. That means the initial power consumption and the double whammy of cooling costs to remove the heat produced by that power will grow apace. Efficiency and reducing the number of units of computation is an overriding requirement to reduce that double whammy.

Hence better performance, or speed is critical. Better scalability is critical, because how a program executes outside the cache hierarchy exposes how it overconsumes energy inside the cache hierarchy to hide its lack of scalability.

Memory-Centric Computing

Memory-centric computing championed by the Gen-Z Consortium and HPE Labs Machine project [Gen-Z, slides] can reduce the size of data centers by allowing hundreds or thousands of nodes to share a vast pool of memory:

- Using byte-addressable instructions to access those enormous shared pools of memory
- Using load-store memory semantics: unbuffered and fetched or paged on demand
- With cache line granularity and processor uncore pre-fetch only limited by memory page size
- Supporting non-volatile memory persistence and erasure coding crash recovery directly in software that is much faster than hardware (like gerasure)
- Supporting fabric atomics to allow non-coherent memory-sharing across nodes at massive scale
- Using memory channel semantics and failure paradigms and not PCI bus/RDMA (2 orders of magnitude slower), requiring erasure coding statistical techniques for posted writes
- Finally, using base + offset addressing for non-coherent memory as opposed to virtual addressing for coherent memory, which cannot scale out to exabytes of memory shared across thousands of nodes within anyone's reasoning or composability paradigm

- All of that allows data sharing in memory across nodes with crash recovery and without the burden of marshalling, buffering, messaging, I/O for persistence, and PCI bus/RDMA handling, simply by using the memory channel which is orders of magnitude faster and has a native unit of granularity orders of magnitude smaller (cache line vs. memory page.)

3. Why not SQL for the DDL/DML, and why C++?

For systems programmers in the 1970s, you coded in assembler and systems languages of many sorts on many, many operating systems. Now there is C++, which compared to those compilers is a robot programmer of immense skill, experience and precision. This trend appears to be accelerating into something wonderful, but not necessarily for human consumption, ease of use, code readability, etc. The robot programmers of the future will likely care much more about deterministic and predictable performance, scale, behavior, fault detection, fail-fast or fail-stop, and consistent crash recovery … than they will care about self-documenting and intuitive interfaces, coding standards and the like.

An SQL front end could probably be put on the current C++ code base described here, but why bother? As soon as you did, you would need the whole factory of offline SQL components to support it: catalog, utilities, etc. Also, any equivalent code written in the dynamic languages (Java, Python, Ruby, Perl, etc.) will be much, much slower than the performance of the current code base. If you had cache line pre-fetch or paging on demand in the mmap of Java or Python, then they might be slow and barely usable for these kinds of systems, but that hasn't been seen. Embracing the future: fast, simplified and pleasing to the C++ robot programmers, can also save the planet and make data centers much smaller, more numerous, more locally low latency and vastly less of an expense.

What is suggested here is a 3-6 order of magnitude improvement in efficiency and time, with some "small matter of software" (SMOS, not the ironic SMOP.) In addition, there is the benefit of a vast increase in speed and scale when using C++ over SQL.

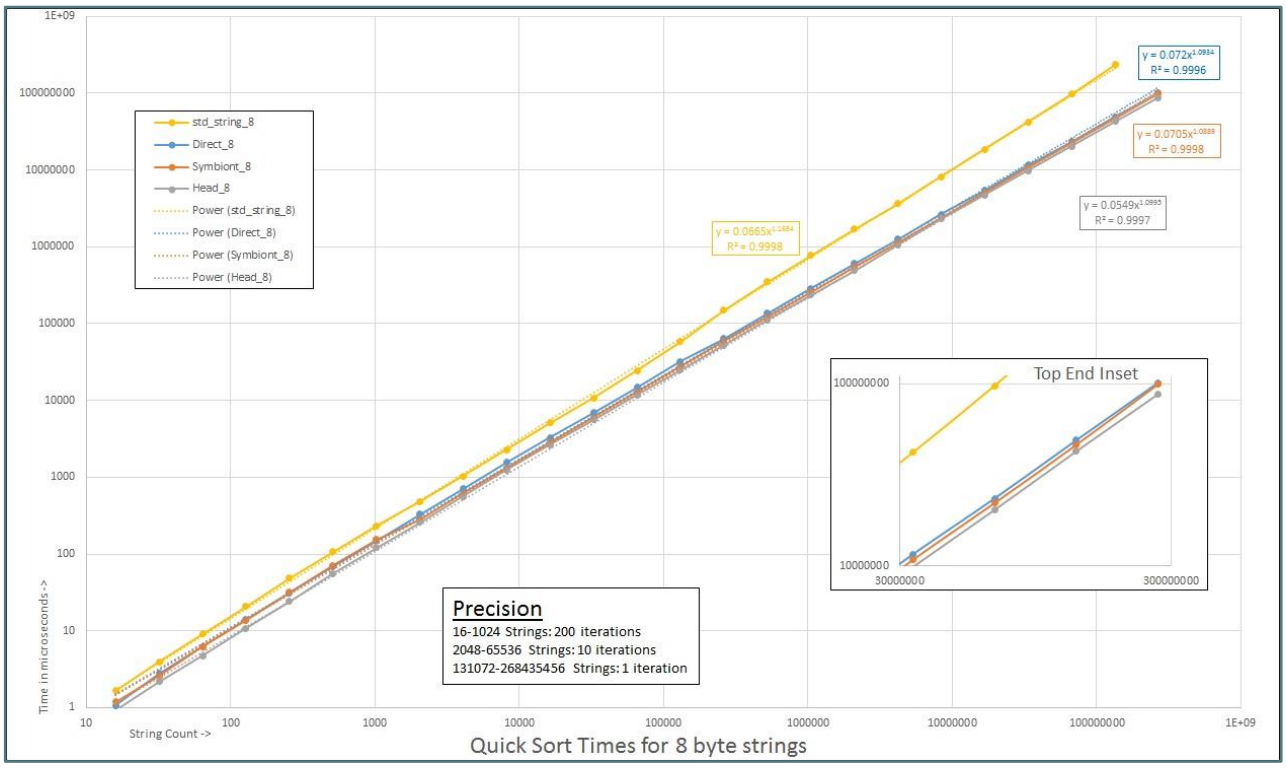
4. SortBench program results: comparing the **Direct**, **Head**, **Symbiont** and std::string classes

Caveat: these are single-threaded sorting comparisons. Multi-threading, distribution, and best sorting algorithm are a separate exercise. Having the fastest strings on the fastest containers makes for the fastest single threaded sorting, independent of the algorithm, distribution and the parallelization. Slow components are not worthy of fast algorithms and excellent distributed parallelization.

SortBench is a program in the Kaldanes GitHub code base:
https://github.com/charlesone/Kaldanes

SortBench is a C++11 Linux console program churning out performance output statistics line-by-line to the console. It does performance analysis using the precision nanosecond clock support from C++11 on Linux. **SortBench** has not yet been run on Windows.

Comparing Short Strings



[The charts are big and dense and will require zooming in to read the superscripts online, it is doubtful they will print well. Log-log plots are exclusively used due to the scale and in some cases, vast differences. So, what looks like a small difference can actually be an order of magnitude.]

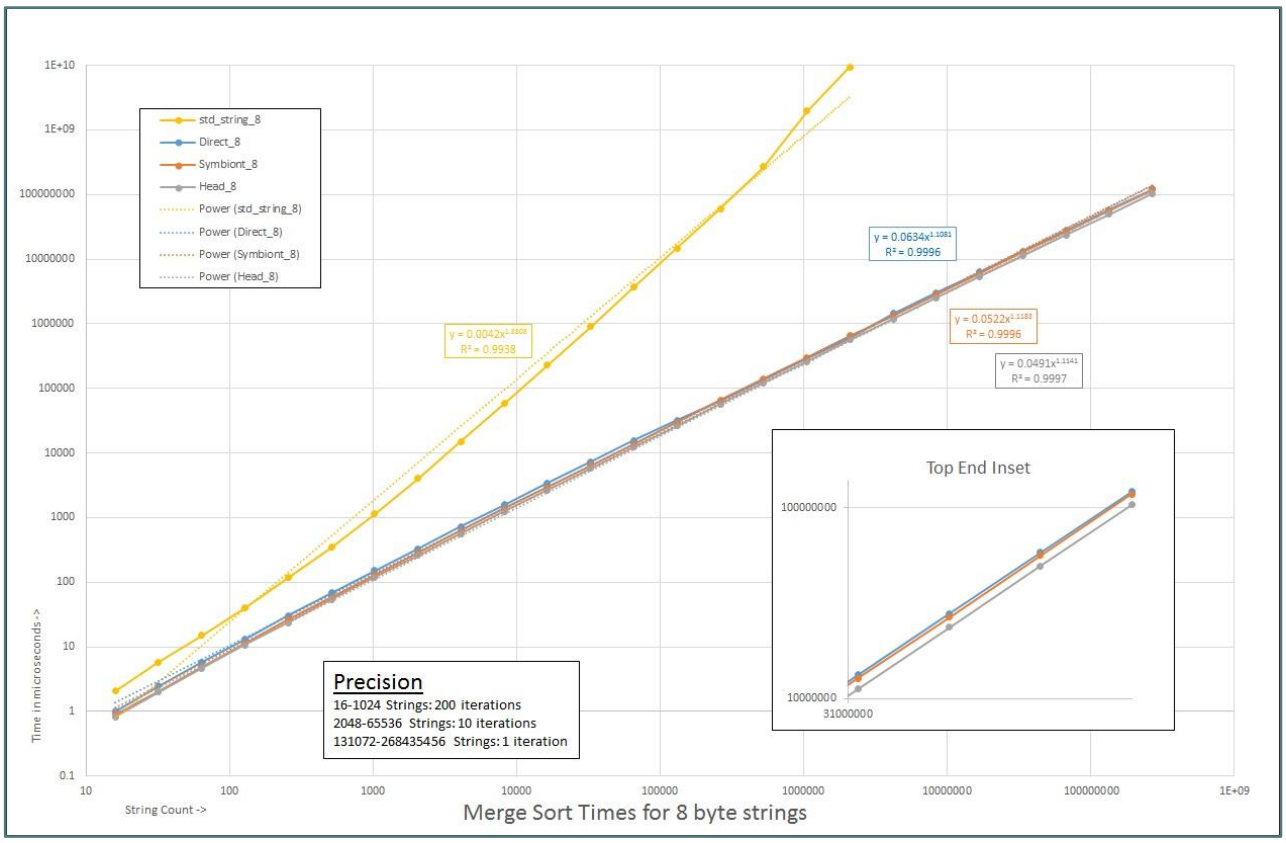
Four types of 8-byte strings are tested against the same template quick sort functions, and next plot, merge sort: (1) std::string class objects are provided by C++ libraries and have the ability to be shared with copy-on-write support for threading, so moving them is by pointer, (2) **Direct** template class strings, which move as a single element during sorting, (3) **Symbiont** template class strings, which are stored in a block with a head and a body, such that during sorting the heads move and the bodies stay in place, and (4) **Head** template class strings which have their heads stored in a separate array, and which declare a **Direct** array internally to store the bodies which don't move.

Looking at the plots above, one might think that for small strings moving them entirely in the case of **Direct** strings would be equivalent to moving the **Head** of the string as a PMNK (poor man's normalized key,) but for some reason the tuning on **Head** by the pmnkSize optional template parameter has made it a tiny bit faster somehow than just moving the small strings. Not clear why.

It was found that there was a lot of jitter in the plot at the low end (fewer strings in the sort,) because the time goes by so quickly. To smooth the plots generally, a greater number of iterations is done for the small string counts. Those figures are quoted in the precision box. The error bars could be computed from that.

The files concerned are located in the code base at https://github.com/charlesone/Kaldanes and can be built by the "`make all`" command (including header files `Direct.h`, `Symbiont.h`, `Head.h`, and `Sorts.h`).

A generically interesting thing is how random strings are generated for the `SortBench` program. The number of calls to the random bits generation is reduced by an average of four across different string lengths. This was done by generating them 64 bits wide at a time into a `char` overlay. No comparisons were done, but it is mighty fast.



The divergence above for std::string was quite a surprise: std::string is quadratic for merge sort, and the new string classes aren't. From inspecting the code, the type-related time complexity difference between quick sort and merge sort is that quick sort uses a single temporary variable of the string type in the inner loop to do three-cornered swaps (left to temp, right to left, temp to right) and the merge sort uses a temporary array in the recursive call to allow two-cornered swaps in and out of the temporary array. Initially the working hypothesis was that the quick sort single temporary item stayed in first level cache or optimized to some registers or a scratch pad in the micro architecture, and the merge sort had cache performance problems, but that would not account for quadratic behavior. Something happened to change that hypothesis.

On one run, **Head** strings became quadratic on merge sort, remaining linear for quick sort (with a much better scale factor than std::string, but hey!), but still linear for **Direct** strings on both sorts. It quickly became obvious that the code had been compiled for debug instead of release, and that prompted a thought: what was the debug vs. release difference between these two classes that might make a difference related to swap arrays vs. a single swap variable in the two sorts?

Direct was constructed as an attempt to support more than one character type as `typename T` (`char`, `signed char`, `unsigned char`, `wchar_t`, `char8_t`, `char16_t`, and `char32_t`), but **Head** was a more complex class and could only be made to just support `char`, because `wchar_t` and

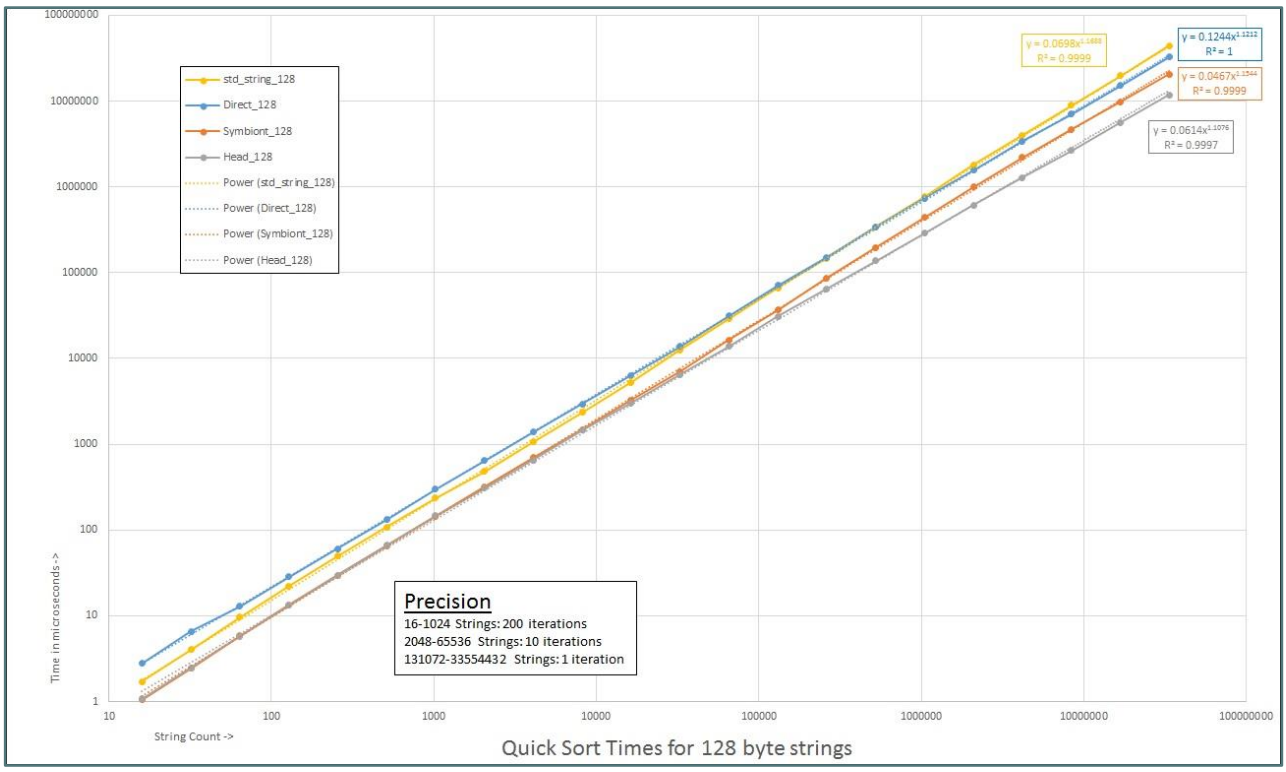
wider char type support for the needed routines was not there. The g++ compiler for the **Head** template class had demanded an empty default constructor for `Head(){}`, so it was supplied. From experience in language and tools at Tandem in the debugger group, it was realized that release code optimization was probably getting rid of that procedure, but retaining it in debug for break points and stepping. That might be the source of quadratic debug performance in Head for merge sort, but how?

It might be in initializing those temporary arrays … and that must be the case for std::string, except that constructor has actual code in the release build precluding being optimized out, and then merge sort is quadratic and that would turn out to be true for any length of std::string. Could this be a problem in string and other structure handling in graph systems and other systems? The std::string class is copy on write, and although that is changing in future releases, but it's not likely that they can keep its vast functionality and have an empty default constructor, so the quadratic problem with temporary arrays may not be fixable, for std::string, for plug-in replacements for std::string or for other classes which do initialization work in the default constructor.

There is no problem with merge sort and the other fundamental types (`int`, `long`, `long long`, `float`, `double`, `long double`), this problem for temporary arrays is isolated to std::string and other classes and types with constructors that are not empty (these would be a large percentage of the advanced C++ code base in the world, it is thought.) You could just use quick sort, but the problem there is that merge sort is so easy to multi-thread and multi-process, because it breaks down the sort into divisions that are sorted separately and the merged and sorted again, and that is just what you need for easily isolating the sub-sorts in threading. (e.g., yellowed videos of merge sorts simultaneously employing all of the tape drives in the data center back in the 1960s.) Also, any data movement in systems using temporary arrays to avoid three-cornered swaps will run into this behavior for any types without empty default constructors.

However, not a problem for **Direct**, **Symbiont** and **Head a**nd any future Kaldanes-style classes.

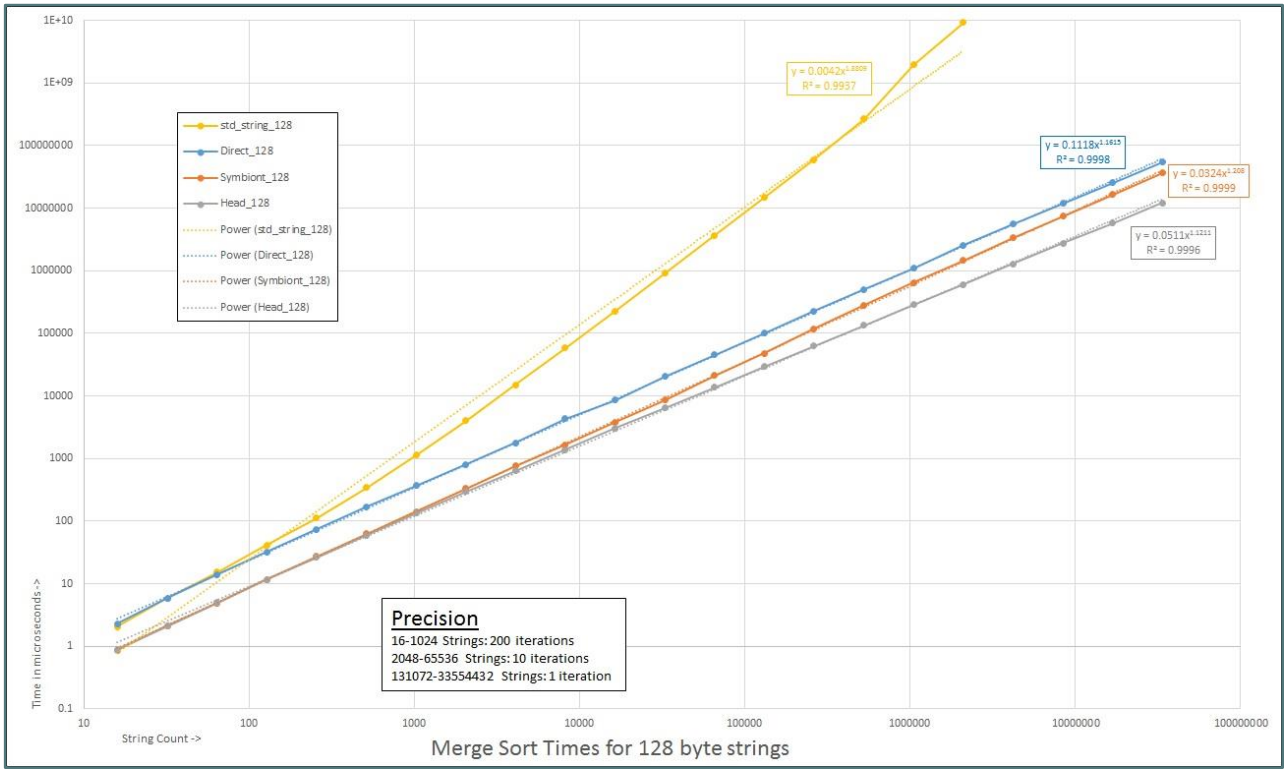
Comparing Medium Strings



128 byte string quick sorts and next, merge sorts involve strings that have their bodies stored across 3 contiguous cache lines, occasionally two. Hence, **Direct** (in blue,) which swaps whole string bodies, should have a disadvantage versus the strings just swapping the PMNK elements (**Symbiont**, **Head**) or a 64-bit pointer (std::string). However, due it is assumed to cache line pre-fetch in the uncore of the processor, **Direct** is close to std::string and later catches up and passes it outside the cache hierarchy (at the 10 MB slab size, or so). 128 bytes is short enough that there is only an occasional traverse of a physical page boundary, which blocks uncore pre-fetch, because the uncore has no notion of virtual address contiguity. The std::string has a pointer and a body in different cache line locales, which should scale better by just swapping pointers, but since it supports copy on write, perhaps there is a counter in the body that must be modified or at least examined on address modification, even if the head of the string were stored with the pointer element for comparisons.

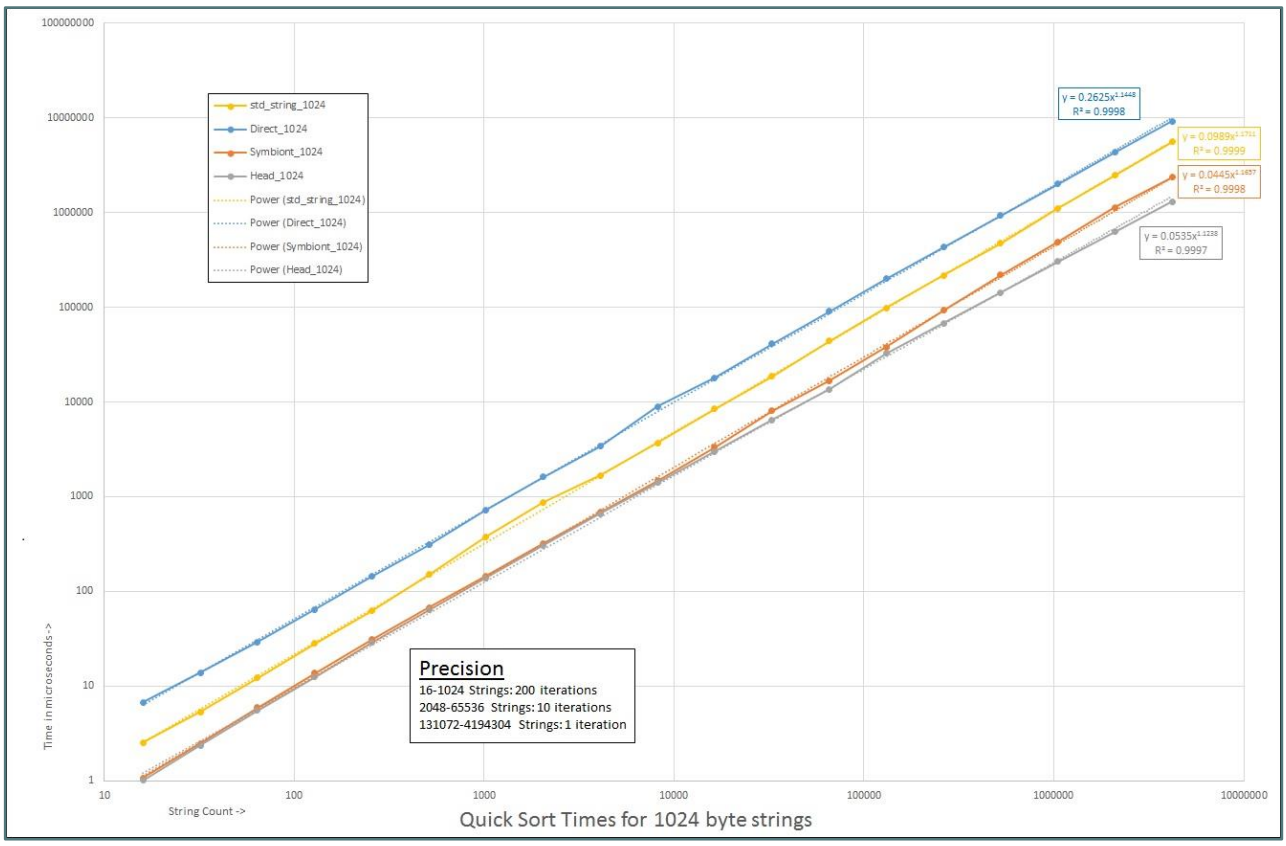
Head is much faster than anything else and is pulling away at scale, outside of the cache hierarchy spatial locality of reference domain. This is due to the superior temporal locality of reference of **Head** over **Symbiont**. **Symbiont** has the heads and string bodies in the same slab and simply sorts the heads, leaving the bodies in place, immutable and immovable: thus after sorting, every head is adjacent to a body that it does not reference and this makes it a fast moving string. However, **Head**

strings have more elements packed into cache lines in the array, and **Symbionts** have intervening bodies taking up space between head elements and needing to be stepped over and thus loses out in performance to **Head** increasingly at scale.

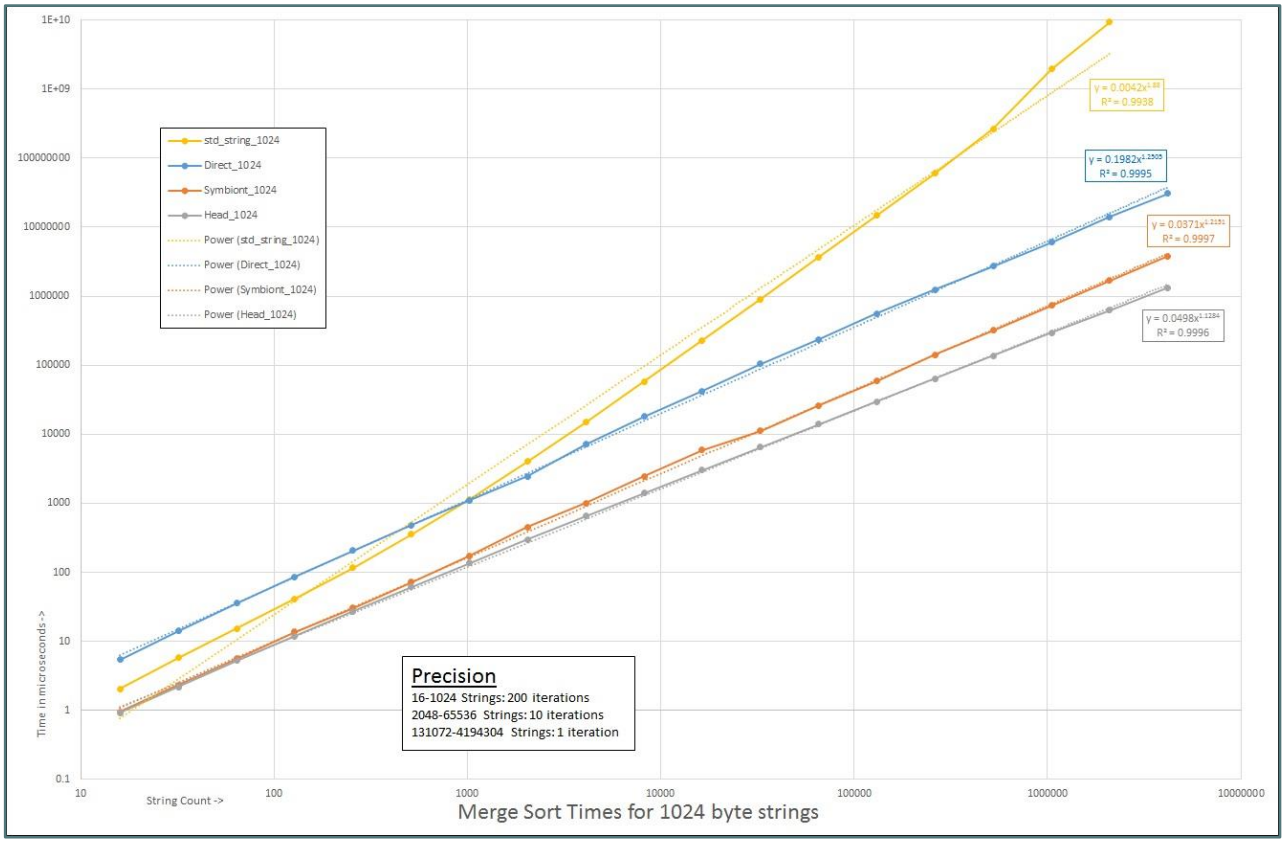


Nothing new here, quadratic problems for std::string predominate and **Head** wins at scale due to superior temporal locality of reference. **Direct** is starting to lose the battle. Notice the bump at the end of the std::string curve at around 60MB or so: the effects of the cache hierarchy from spatial locality of reference are dwindling and it's taking off on a new vector. In the next iteration (not shown) std::string runs out of memory and dies, but that memory is not for the string bodies, it's some kind of metadata, possibly the fine-grained allocation overhead that the other strings aren't doing. A bigger memory system (than 15GB) is needed to characterize that.

Comparing Long Strings



For 1024 byte strings on quick and next, on merge sorts, **Direct** has lost the battle: swapping arounds 1024 byte strings is no way to sort them. **Symbiont** matches **Head**, until exiting the cache hierarchy spatial locality of reference domain, after that **Head** takes off on a lower trajectory.

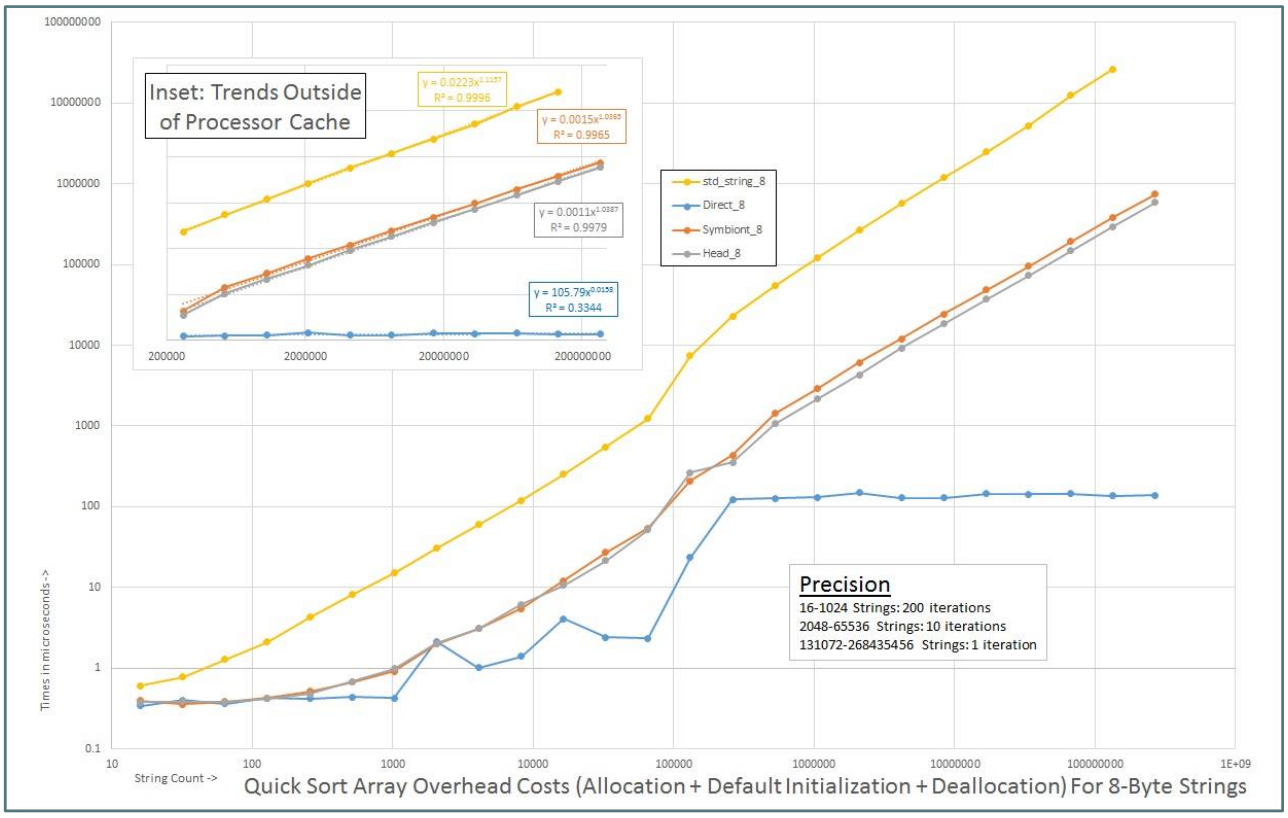


This slide simply follows the previous trends. The name Kaldane is an antique pulp Sci-Fi reference from the 1920s and 1930s (heads move, bodies stay): http://www.catspawdynamics.com/images/gino_d%27achille_5-the_chessmen_of_mars.jpg or google image search the word "Kaldane", or https://en.wikipedia.org/wiki/Kaldane

Hence, the Kaldanes Rule is “Sorting heads is faster than sorting bodies.”

In the case of **Symbiont**, the bodies are always wearing the wrong head, just as in pulp fiction. In the case of **Head**, sorting them by themselves in their spider warrens is fastest. 1024 byte string operations are the most important to what is coming further later, because rows in a relational database can easily get up to that length, or much longer.

Comparing Allocation, Default Initialization and Deallocation Overhead Costs (ADIDOC)

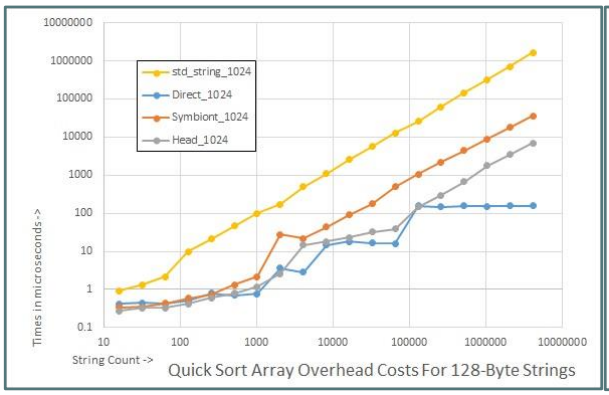


SortBench also keeps statistics (with the precision nanosecond timer) on the overhead costs for allocating, default initialization and subsequent deallocation (call that ADIDOC) of these big data structures. The structure of the graph in the region of 2048 to 131,072 for **Direct** strings (in blue) is most interesting. Most of that is run with 10 iterations, so that structure is repeating and likely reflecting something very real. It would appear that there is a spike in overhead for ADIDOC every time the data chunk size exceeds a level of the cache hierarchy. Also interesting is what happens to **Direct** ADIDOC at 131,072 strings and beyond, it is flat. This is the signature of the operating system slab allocator: 106 microseconds per big allocation with a constant cost unrelated to size. Why are the slab allocated **Head** and **Symbiont** strings not getting that same flat performance?

It's because the PMNK element contains the K-value, the array index, for that array element and that had to be written at allocation time, so the **Symbiont** heads and the **Head** data structures can be sorted during the sort test. Allocation (ADIDOC expense) and loading the arrays with random strings are done outside of the sort tests, of course: so neither random generation times nor the ADIDOC expense will affect sort times in **SortBench**. For **Symbiont** and **Head**, the K-value is preset, and the PMNK head of the string is written afterwards when the table is loaded with random strings. So that cost is linear, but less than a second for 2GB of string data. In addition, an

attempt was made in SortBench to pre allocate the memory management data structures for the scale of the memory that was to be incurred for the test against all four string classes. In a hot system, where long running programs do big things, the memory management table allocation overhead should not be a factor in performance (IMHO).

Note that fine-grained allocation overhead for std::string is a serious loser: somewhat greater than linear with a larger scale factor, somewhere about 30 seconds for 1GB of string data. At 2GB, the std::string test dies, out of memory on a 15GB system: so there is some suspicious space complexity there in the allocator. Nothing else was running when these tests were done. Here are the charts for 128 and 1024 string ADIDOC expense. Only the quick sort ADIDOC numbers are shown in all of these slides, because they are the same as merge sort.



The Sort Results Are a Guide

Hence, **Direct** seems to be perfectly bad for sorting, but perfectly good for allocating enormous tables, so it looks like a candidate for relational database tables. And **Head** seems to be the best for making indexes for those tables.

In the future, when these tables and whole databases are slab allocated using mmap in their finalized form, there will effectively be much reduced ADIDOC for consumers of prebuilt databases: only suffering the slab allocator 106 microsecond cost and the mmap `MAP_POPULATE` cost.

It is the table, index, and memoized query mmapped slab producers that will pay the ADIDOC expense.

Consumers would then receive an mmapped slab database whose logical form is its physical form, through base + offset addressing with memoized joins of any number of tables of any size effectively for free, as shall be shown later.

5. SortDemo program

Whereas the SortBench program can take days to execute a run of benchmark tests, **SortDemo** executes in much less time. It gives a warning when the std::string merge sort tests for a million strings are about to run (they take a half-hour each), and it is thought that most users will get the picture and hit ctrl-c at that point. Unlike **SortBench**, which generates new random strings for each test, SortDemo copies the identical set of random strings into all four string data structures, for apples to apples comparison. The number of compares and swaps taken by the quick sort and merge sort are measured, and these must be the same across all four string classes for the same number of same length strings, so compares and swaps are printed out as well. The results of one such run of SortDemo are in the Appendix.

6. SortQATest program

SortQATest was taken from the source for **SortDemo**, with the std::strings removed (they take too long.) **SortQATest** throws an exception if the three string classes produce different numbers of compares and swaps from sorting the identical string arrays using the identical code. Also, since merge sort only uses the <= comparison, and quick sort only uses the < and the > comparisons on the actual data (not integer indexes), for coverage it was decided to walk the sorted arrays with all possible Boolean combinations to test all of the comparison functions, comparing both the previous and successive array element against the current one, for consistency against the sorted random data, and this caught some subtle bugs. The `pmnk` (poor man's normalized key) compare logic is complex and highly optimized to never, ever go to compare the tail string if the result can be deduced from the `pmnk` head string. Comparisons with self, or a pivot copy of self, reduce to K-value equality. **SortQATest** runs forever and the output looks like SortDemo, so that output is not included in Appendix A.

7. Building a relational database system using slab arrays of strings

The three relational database demo programs (**TableDemo**, **JoinDemo**, and **BigJoinDemo**) are constructed with two sections before the `main()` function: (1) a Static Metadata Section (SMS) laid out, in an absolutely order critical way, containing everything (classes, types, static data, const and constexpr) that is needed for generic programming (compile-time) of the relational database system, and (2) an Active Program Section (APS) laid out in whatever order that is desired, in this

case containing performance analysis declarations and functions, data loading and printing functions, and a runtime checker to guarantee the consistency of columns versus tables in the layout in the SMS.

This is not in any way the normal organization of C++ programs up to this point in history. After much evolution and refactoring, this relational database system C++ program morphology may change yet again, but almost certainly not reverting back to the old standard of program design (IMHO.)

All of the elements of the `main()` function, are executed within an anonymous exception handler, that prints out the ugly extended name of the exception with the readable name at the end, and that works in release code for bug analysis during performance testing.

These demo programs use automatic variables allocated on the stack to create the large slab database objects in the `main()` procedure, and those would automatically disappear if they were allocated in a procedure and then that procedure returns. That is what mmap is for: to manage virtual address space against a large memory system (hopefully a Gen-Z system, or just a huge chunk of DRAM or 3D_XPoint.)

"Those who code, code bugs" – Jimbo Lyon, "It is fine to be on the leading edge of technology, but avoid the bleeding edge of technology." from *Why Do Computers Stop and What Can Be Done About It?* - Tandem Technical Report TR-85.7 Jim Gray.

Of course you cannot avoid the bugs or the bleeding edge, if you want to accomplish anything of significance. When issues are mentioned in text about the g++ compiler (gcc 4.8.5) that don't work or which require workarounds, these are not complaints. The fact that software can be found that works at all in complex new areas of programming, is a source of deepest relief and gratitude.

a. **RowString** class (variation of **Direct**) in simple arrays for tables

Both the **RowString** template class and its accessing **IndexString** template class share the same 5 basic template parameters and operate on character `<typename T>` strings, although the current implementation is unabashedly `char` typed and won't work for other string types (yet.) Both classes are completely base + offset, accessible under virtual addressing, and the objects are portable individually as elements or as slabs: for ease of replication. The variable format ***is*** the log record format. The current demo code base does not use mmap for accessing these objects, but that will work just fine in future.

RowString variables are variable-length (with an upper bound) collections of null-terminated byte strings that have their bytes moved as a single body when they are swapped as in sorting, not just moving their pointers (following the Sort Benchmark rules.) **RowString** has "value-string semantics" as Stroustrup defined for his String type (*The C++11 Programming Language*, 4th edition, chapter 19.3).

Imitating the b-tree block there is an offset, in the front of the **RowString** object, to the column count and column offsets at the end. This allows quick loading from simple .csv files (these "comma-separated-value" files are loaded using one of three member functions named `assignColumns`) without commented or escape characters, by copying in each string after the initial offset and simply converting the commas to nulls as you build the column count and offsets. That's very fast for loading.

There are three assignment functions provided for moving `char` strings into a **RowString**:

- `assign (const char* str)`: which is for a simple one column **RowString**.
- `assign (const char* str, std::size_t size)`: for which the user has preformatted the **RowString** correctly with columns, nulls, counters and offsets.
- `assignColumns (const char* str, const char delimiter = ',')`: described in the paragraph above.

An effort is made to provide as many `constexpr` member functions as possible, to supply type information at compile time for generic programming. That effort pays off during the variadic template parameter pack recursion later (**QueryPlan** and **JoinedRow**), which is effectively an exercise in manufacturing arrays of constants and a little bit of code from typed parameter constants at compile time.

Both **RowString** and **IndexString** make extensive use of `static` global constants and the `enum class` constants `Table` (identifying all the table arrays of **RowString**) and `Column` (identifying all the index arrays of **IndexString**.)

Accessor Member Functions and Dropping the Anchor

Two accessor functions were found to be necessary on the **RowString** class, which are used heavily in join operations:

- `char* columnStr(Column columnEnum)`: This returns a char pointer into the column inside the row element being operated on.
- `rowType row()`: This returns a typed pointer on the row itself.

After supplying the basic comparison operators for the **RowString** class (which are there for convenience, since tables are not for sorting, that's what indexes are for) and the input and output `ostream` operators `>>` and `<<`, a member function called `dropAnchor(RowString rowArr[], std::size_t size)` is specified, for the purpose of storing the "base" of the base + offset for the table array (slab) of **RowString** elements and its size.

This anchor allows the **RowString** element of an array (slab) of **RowString** elements to know information about the array that is not passed or is lost in successive calls or use of `constexpr`. That allows every **RowString** element to know where it fits in the original table structure, even when it is copied out by reference as a (PMNK, K-value) element (where the K-value is the array index of the original element) in an **IndexString** into a pivot element, or temporary variable or array during sorting. The anchors, where they are required in these classes, are the source of all base + offset referencing by other classes in this relational database system.

b. **IndexString** class (variation of **Head**) in simple arrays for indexes on its **RowString** friend class

As stated before, the **IndexString** class starts with the identical template parameters as the **RowString** class table that it creates an index for. In addition, **IndexString** has a separate tuning optional template parameter named `pmnkSize`, which defaults to 7, the sweet spot for well-behaved strings. Both IndexString and RowString carry their internals in a `struct`, because that makes copying and sizing operations more convenient, and the sizes of those are in increments of 4 bytes from C++.

Poor Man's Normalized Keys (`pmnk`)

The **IndexString** `struct` contains the 4-byte K-value as an index into the array of **IndexString** objects at the front, and then the `char pmnk[pmnkSize + 1]` null terminated string that is the head, of all of the column strings inside the **RowString** that are being indexed. That means that `pmnkSize` should logically have values of 3, 7, 11, 15… for strings without lots of repeating characters in the front (e.g., " Joseph Cotton") and that have a somewhat random distribution, 7 is a good average size and that keeps the system from having to jump to the rest of the string kept in the **RowString** column, because the `pmnk` kept in the **IndexString** was too short to satisfy some comparison.

As is stated in Goetz Graefe's paper: *B-tree indexes and CPU Caches*, Goetz Graefe and Per-Ake Larson: IEEE Proceedings of the 17th International Conference on Data Engineering Section 6.1, "Poor Man's Normalized Keys", p. 355:

> *This technique works well only if common key prefixes are truncated before the poor man's normalized keys are extracted. In that case, a small poor man's normalized key (e.g., 2 bytes) may be sufficient to avoid most cache faults on the full records; if prefix truncation is not used, even a fairly large poor man's normalized key may not be very effective, in particular in B-tree leaves.*

It might be cautiously suggested that indexes maintained on data requiring a lot of prefix truncation might have lower value in any case: mining low-grade ore is bound to be less satisfying than mining high-grade ore.

Friend Class, Accessor Member Functions and Dropping the Anchor

At the top of the **IndexString** class, the template parameters in common with the **RowString** class template fully-defined type, are used to declare **RowString** as a `friend` class. That allows access to all of the **RowString** table internals that the IndexString index is an accessor for. You could have gotten some limited access with a `derived` type, but then you would be carrying around the bodies of the **RowString** in the **IndexString** elements, so the `friend` declaration is a saving grace.

To get access to the table row from the index, several accessor functions are added:

- `row()`: typed access to the actual row table element that the index element is pointing to.
- `rowAnchor()`: typed access to the base of the table, which the index is pointing to an element of, such that it can be indexed separately (e.g., join linkage on the "from" table).
- `c_str()`: char pointer access into the column of the table row that the index element is pointing to.

All of the comparison operators are supported, both against `char` strings and other **IndexString** elements (of the identical type), this allows you to compare different columns within the table, and columns across different tables. There is an `ostream<<` output function that outputs the **RowString** table element that the **IndexString** element is pointing to, but there is no input function: that is supplied by a member function called `copyKey()` that is only called by the anchor routine below.

The anchor routine for an **IndexString** index can only be called after the **RowString** table, that it is indexing, has been fully loaded by one of the three assign member functions, and then has had its

own anchor dropped. Calling `dropAnchorCopyKeysSortIndex(IndexString indexArr[], std::size_t size)` then does the eponymous three things, returning a working index.

Any column of a table can be indexed at any time after the table is built and anchored, but to repeat the critical point, these are automatic variables allocated on the stack in the demo programs, and they will disappear if they are allocated in a procedure and that procedure returns. That is what mmap is for: to manage virtual address space against a large memory system.

c. **RelationVector** class holds a from-column and a to-column as **IndexString** class objects

In an SQL language query that joins tables, the connections between tables are reflected in a where clause or a join statement with foreign keys such as:

```
SELECT table1.column1, table2.column4, table1.column3
FROM table1
WHERE table1.column3 = table2.column7;
```

Then the offline tools, in particular the SQL compiler and optimizer, make informed choices as to what that statement means at runtime. For more complex joins, there might be no connection between the parts of the join: for example, two sets of where clauses of four completely separate tables will have no linkage for the compiler to generate a query from: that will generate an SQL error, unless there is some rule to rectify it.

In the relational database system presented in the current code base, it was decided to use **RelationVector** items as template parameters, which contain (from-column, to-column) **IndexString** pairs on one or two tables with a vectored direction, and the order of those relation vectors in a set to deduce the join method for that query.

RelationVector Class Objects Soft-Define `JoinedRowStruct` Internals

In processing ordered sets of relation vectors as a join, the first relation vector establishes two columns on those one or two tables as a basis, yielding a first from-table and a first to-table and then produces a range query linking the first from-column and the first to-column introducing a first to-table range context of row values. After that, the connection between successive relation vectors defines a range query linking the from-column and the to-column of each additional relation vector: that introduces a new to-table range context of row values, and potentially new to-tables to the join.

Hence, the ordered relationship between relation vectors creates a large joined table consisting of the first from-table, the first to-table and all successive to-tables from the ordered set of relation vectors. This process mirrors the programming by hand of nested-loop equijoins precisely, and the top-down nature of nested-loop equijoins guides the creation of the linkage rule for them.

That rule is simple in concept and execution: the from-table in each successive relation vector, after the first, must pre-exist in the previous relation vectors as a to-table or a from-table, in that order, going in reverse. The code for initializing this (`constexpr void initJoin()`) is relatively simple, only made complex by the nature of variadic template parameter pack rules and the recursion methods for termination of recursion making coding more complex.

As we shall see below, an N-dimensional set of **RelationVector** types passed as a parameter pack into the **QueryPlan** class and then subsequently into the **JoinedRow** class, will soft-define, by the **RelationVector** contained types and order, an N+1-dimensional `JoinedRowStruct`, which is produced by the **QueryPlan** class and becomes the basis of the internal data of the **JoinedRow** class objects in arrays of relational join query output.

That soft-defined quality means that routines creating or processing any `JoinedRowStruct` will always involve compile time recursion across the parameter pack of **RelationVector** types passed in a parameters to the **QueryPlan** class and the **JoinedRow** class. And as complicated as that gets, for performance purposes, it's a very, very, very good thing to have the compiler do the work and to banish that overhead cost from the runtime of the user.

Static Polymorphism Using `reinterpret_cast` of `void` Pointers

Static polymorphism became a necessity for the **RelationVector**, which contains the "from" and "to" **IndexString** pointers, because those objects don't exist, except as types, until runtime initialization of the relation vector objects from the actual `dropAnchorCopyKeysSortIndex`-created indexes. This makes compiling the various accessing code, such as placing a parameter pack of relation vectors into a tuple and then accessing those objects in the parameter pack recursion for things like join, problematic.

The solution was to leave the pointers uninitialized and then in the accessor routines `fromIndex()` and `toIndex()`, to do a conditional test before returning them, incurring a one instruction permanent cost of a register test for zero. If the test fails then the type is used in a `fixFromPtr()` or `fixToPtr()` procedure via `reinterpret_cast` of the void pointer stored in a static global array at runtime during initialization. [`auto` arrays would be very useful here, were such a thing made legal.] The use of `constexpr` for those member functions had to be abandoned, because

`constexpr` in C++11 discards type information on the pointers being returned in both sets of procedures, so `inline` had to be used.

d. A heavyweight relational **QueryPlan** class is defined by a variadic template parameter pack of ordered **RelationVector** class objects and creates a shared tuple of them

The strategy for the **QueryPlan** class was that it would bear the burden of future metadata and join query initialization and processing required by the extremely lightweight **JoinedRow** class. In the current code base that burden is middling, but that will grow as more features are added. Of course that meant it had to precede the **JoinedRow** class in the declaration order, and yet be deeply coupled to it by a lot of machinery. So, both classes are located in the same header JoinedRow.h in the obvious order and share static globals for the handovers back and forth.

Variadic Class Template Parameter Pack Compile-Time Recursion

For a host of reasons: the need for true static polymorphism at scale, the complexity and C++11 deficiency of handling for templates with optional parameters, problems with the C++ optimizer in complex variadic recursion, the necessity of using static globals, the problems of using constexpr in a template class with template static data, and more … it was decided that the only stable way that would scale to deeper recursion in the compiler, was to use the SFINAE – termination method of recursion.

It was also found that declaring local variables inside deeper levels of recursion made the compiler perform in an unstable manner, e.g., if you declare a local variable inside a recursion on a sufficiently complex variadic template parameter pack and don't immediately use it, the compiler spins madly spitting out recursion context messages and then does not produce the compiler warning on the unused variable. An error or warning without a message can be quite difficult to diagnose!

Add to that the fact that you cannot use the debugger or even print statements when the code doesn't compile for an unknown reason. Sad to say, even when generic programming code compiles, when it doesn't work the way it should, you cannot really debug the code that is executing in recursion in the compiler, what you get is a debug session emulating the runtime aspects of that. And print statements come out in strange orders from ostream in generic recursion recursive routines, and it seems impossible to rectify that disorder.

Hence, the usage of an array of frame structures holding indexed locals at each level of recursion is a necessity: and then the strategy is to resuscitate an archaic driver technique. Then the problem arises, how deep are you in the recursion?

This is solved by using the two layers of parameter packs, the template class parameter pack `template<typename... classPack>` on the recursion-SFINAE-termination version of `join()` and the pair of template parameters on the working recursion procedure `template<typename packFirst, typename... packRemaining>` version of `join()`. Here's a code snippet for discussion:

```
template<typename... packRemaining>
constexpr typename std::enable_if<sizeof...(packRemaining) == 0>::type join() {}

template<typename packFirst, typename... packRemaining>
void join()
{
        const std::size_t packDepth = sizeof...(classPack) - sizeof...(packRemaining) -1;
        const std::size_t maxPackDepth = sizeof...(classPack) - 1;
        if (debugTrace) cout << endl << "join() packDepth = " << packDepth << endl << endl;

        // these are compile-time only, no runtime cost
        typedef decltype(((packFirst*)0)->r.from) fromType;
        const Column fromColumn = ((fromType)0)->enumColumn();
        typedef decltype(((packFirst*)0)->r.to) toType;

        if (packDepth == 0) arrayOffset = 0;

        if (packDepth > maxPackDepth)
            throw Variadic_Parameter_Pack_Logic_Failed();
            // constants are seriously out of whack.
        else
        {
            if (!frame[packDepth].initialized) throw Initialization_Failure();
. . .
```

The value `packDepth` is a constant calculated from the constant size of `classPack` (a set of **RelationVector** types) by subtracting the constant size of `packRemaining`, which is greater as the compiler recurses deeper. At runtime you can think of this recursion as an array of types and constants generated during the compile with a constant `packDepth` as the depth gauge (it only varies in the compiler.)

Similarly the section of declarations below that line are the `typedef`s and a constant which vary at different levels as the `packFirst` element of the parameter pack of types from `classPack` get peeled away one by one until `packRemaining` has zero value. Even though the **RelationVector** pointers for `packFirst`, `fromType` and `toType` are being typecast onto zero values, the `decltype` for the first and `constexpr` member functions used in the others are still `const`: although these look like variables, they're not.

To Reenter (After Interrupting), or Not to Reenter (Just Hang Out)

The problem of shipping data out of a software mechanism is faced in implementing the `void join()` recursion. The query plan is completed by `constexpr void initJoin()`, which sets up the variable frame array (which became necessary at this scale from the intolerance to local variables in variadic template recursion) and a plan to imitate driver logic was chosen.

Drivers, which need to be interruptible by other drivers of higher priority, used to save their state by saving their registers, scratchpads, and interrupt stack somewhere, or simply by pushing down the context via some mechanism supported by the CISC architecture, and now in RISC it is assumed this must be done by hand. It was hoped that keeping all these recursion locals in a static global array of frames might allow cheating on that to work, and then in typical server style, the **QueryPlan** class join having found a valid single joined row, or a rowset of joined rows from the join query, would then interrupt itself and return the data to the **JoinedRow** class caller, and thus external iteration would produce a completed array of joined rows satisfying the query.

That effort failed, because the preempted state could not adequately be restored by any reentrant coding, the rank impossibility of which should probably be a lemma or even an axiom for C++ development. Upon rethinking the strategy, it was noticed that all the possible valid joined rows were produced at the bottom of the recursion (`packDepth == maxPackDepth`) and then they could be drained there, if the output array length was known, which it is. Additionally, without preemption or reentrancy, the code is faster by not having to trek all the way in and out for every joined row or joined rowset, iterating across the giant outer loop. If a remote interface replying rowsets is needed later, a counting semaphore will probably guard the output array in the inner loop while it is copied out by a message replying thread, while retaining the maximum velocity for the inner loop.

Having made that work, and with an incomplete `friend ostream& operator<< (ostream &os, const QueryPlan& rhs)` for the **QueryPlan** class (nothing much to print in the query plan, for now,) `join()` is complete. [There is a lot of dead code in JoinedRow.h that is retained for future refactoring and extension of the current code base.]

e. A lightweight relational **JoinedRow** class is defined by the identical template class parameters as its friend class **QueryPlan** and is coupled to that plan at runtime

JoinedRow class objects are microscopic (a 4-byte `int` per table participating in the join) compared to the table and index metadata and data that they type-infer and pointer-reference, on every other column and index on those columns in the relational database tables.

In the same way that tables of **RowString** class objects should not be sorted and that multiple indexes of **IndexString** class objects on those tables are intended for sorting, it is also true that arrays of **JoinedRow** class objects should not be sorted and that indexes of **JoinedIndex** class objects on those arrays are intended for sorting. However, there are no **JoinedIndex** objects yet in the current code base, because a hack was sufficient to get the demo programs working. That hack reveals the method by which the **JoinedIndex** class will be constructed in the future. Note that the ordering of the joined row on an existing index on a table participating in the join requires no storage of an index string in the joined row element, even to allow range queries on the joined row array: all the indexes can be borrowed by inference. Hence, **JoinedIndex** objects will not contain strings, only a single K-value array offset and the type inference to an index element or table row containing the string. Very, very small indexes indeed.

Since the only indexing on the joins and memoized joins (described below) that was needed for the demo programs was indexing on the from-column of the first **RelationVector** in the common parameter pack of the **QueryPlan** class and the **JoinedRow** class, the shortcut used was to add comparison operators to the **JoinedRow** class that compared against that from-column text. Then by sorting the array of joined row objects after completion of any query (these are tiny objects, even the seven-table memoized joined row object in **BigJoinDemo** is only 28 bytes), then that array can be accessed directly in sorted order with a range query. The difference in performance incurred in accessing an index on the joined row query output will ultimately be nil, because they both use the tuple for referencing data and metadata in the database. The difference in using an index on memoized joined rows versus sorting the memoized joined row output ***will not be nil*** as the number of tables in the join grows, or as the table sizes get enormous. Hence, tiny indexes on joined rows will be needed later.

After that there is a `constexpr void dropAnchor(queryPlanType* queryPlan)` member function to connect up the joined row and its associated query plan, and a `friend ostream& operator<< (ostream &os, const JoinedRow& rhs)` member function for printing out the rows of the tables being joined in this one **JoinedRow** class object.

Normal joins are launched with a single `char` string value as the key, which is used in a range query on the from-index of the first relation vector in the template parameter pack to the **QueryPlan** class (and identically, to the **JoinedRow** class.) That seeds the nested loop join query with one or more rows from the zeroeth table soft-defined from the template parameter pack of input relation vectors to the class, into the joined row successive K-values.

Memoized Joins

Memoized joins are created by using an asterisk `char` string as the input to the nested loop join ("*".) That means that the input rows seeding the nested loop join algorithm, are *every row* of that first from-index. This query will produce every possible valid joined row from the database, sorted (for now) by the from-index of the first relation vector in the template parameter pack to the **QueryPlan** class, by soft-definition.

This allows that a range query on that memoized join output will produce the same output that running the join again would produce, but only for the cost of one range query and a sequential scan of the memoized joined row output. That is very fast, and is the reason that if you ignore the data processing of those output joined rows, they are made available in sequential order for the cost of a single range query, and do not have to be moved somewhere. Similarly, once joined rows are indexed, that will be true sequentially, for every column in every table in the joined row output that has an index for that column.

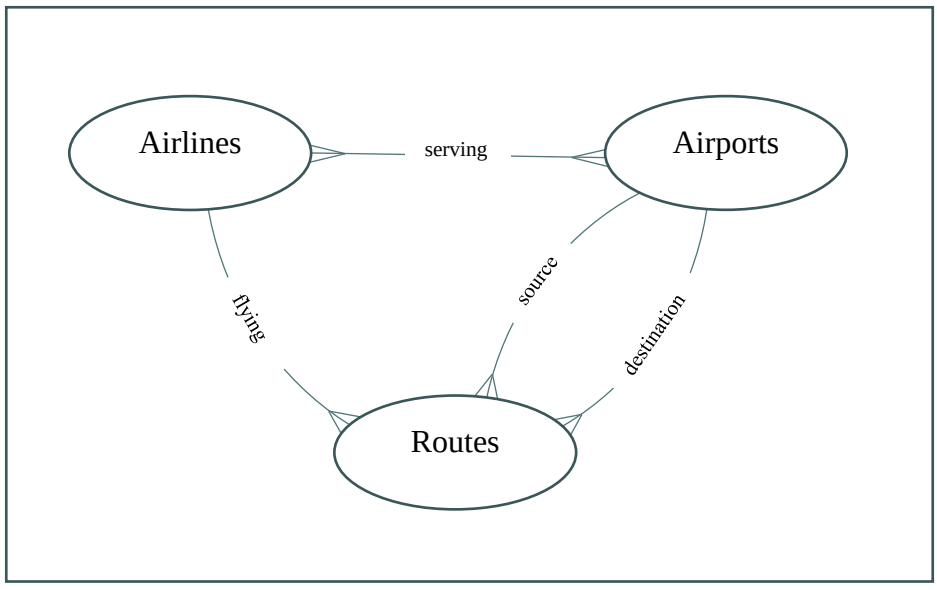
At that point all joins will be effectively free of runtime cost, independent of the number of tables participating, or even the size of the tables and the size of the output. Range queries using two binary searches have an average cost of approximately $2(\log_2(n)-1)$ for consistent databases (because in practice range queries rarely fail in joins in real databases), where n is the number of the output joined rows. The current code base uses two binary searches on the lower and upper range barriers, the second of which could be optimized for small output sets, but that's harder to model than $2(\log_2(n)-1$. Hence, that optimization was forgone for now, but since you only do one range query to get sequential access to the results, the time complexity for memoized joins is still effectively constant time: access cost being much cheaper than processing the contiguous array of results. With some optimization, possibly 70% could be gained, but since that 50% is in first level cache on the second pass through the binary search, it may not amount to much in practice.

8.TableDemo program

As features were added to this nascent relational database system, the **TableDemo** program exposes that history and it still serves as an eyeball check to see if the latest changes have added bugs breaking earlier coding.

The OpenFlights.org Air Routes Database

The database used was accessed from the database used by OpenFlights.org. The files contain data on all the air routes, airports, and airlines as of 2009, not maintained since. At that time there were 65,612 valid air routes (with valid source and destination airports, and airlines.) It's a small database, only 3.5 MB, but you can do big joins on a small database, by folding into the same table more than once. Alluding to the relatively small size of database used in the demo programs, the argument for whether the current code base will scale to enormous size, has three parts: (1) the index sorts are linear time complexity relative to the count of rows while using a small `pmnk` index element, (2) the nested loop joins are linear time complexity relative to the number of output joined rows, and also linear relative to the number of range queries, which have a practical average $2(\log_2(n)-1)$ time complexity for consistent databases, where n is the number of table rows, and (3) the memoized join time complexity is effectively constant time. If you think of the compile cost as including the memoized join cost, then compiling the relational database queries only occurs one time per generation of the database.



At OpenFlights.org, the data is stored in .dat files, those were converted to .csv files by removing quoted strings and replacing commas in those quoted strings with semicolons to satisfy the limited .csv input functioning of the current code base. Everything is #-commented (R .csv style) in the resulting .csv files presented in the code base.

The relations between the demo program tables are (1) many airlines serve many airports [m:m], (2) routes are flown by one airline + airlines serve many routes [1:m], and (3 and 4) routes have one source and destination airport + airports have many source routes [1:m] and many destination routes [1:m].

Once again, the TableDemo program, as all the demo programs do, has two sections before the `main()` function: (1) a Static Metadata Section (SMS) laid out in an absolutely order critical way containing everything (classes, types, static data, const and constexpr) that is needed for generic programming (compile-time) of the relational database system, and (2) an Active Program Section (APS) laid out in whatever order that is desired, in this case containing performance analysis declarations and functions, data loading and printing functions, and a runtime checker to partially guarantee the consistency of columns versus tables in the layout in the SMS.

All of the elements of the `main()` function, are executed within an anonymous exception handler, that prints out the ugly extended name of the exception with the readable name at the end, and that also works in release code for bug analysis during performance testing.

TableDemo keeps track of the time consumed by operations and printing those outputs separately, using the nanosecond clock facility of C++11 on Linux. Initially it builds the database:

1. The three tables are allocated and loaded from .csv files and then anchored into RowString arrays.
2. Space for the ten indexes on those three tables is allocated.
3. The table source anchor is copied into the indexes, the index keys are copied into the indexes from the tables, and the indexes are sorted.

At this point the database is up and ready to query: for the three relational database system demos (**TableDemo**, **JoinDemo** and **BigJoinDemo**,) the average time to produce the database and bring it up from scratch using input .csv files was 118.841 ms, less than $1/8^{th}$ of a second.

First, **TableDemo** uses a point query lambda to print out the 5 rows centered on a specific airport that exists, and one that doesn't exist, so that it can be shown where it would fit in the order of airports. Failed binary searches can be one off the correct location, high or low. That small error needs to be corrected for after a range query boundary value binary string search which almost always will fail, but be within one of the high or low edge of the range where the boundary should have been.

Second, a range query lambda is executed on the airlines index on the routes table, printing all the 378 valid Allegiant Airlines routes. Those are printed out, but that list (many pages) is abbreviated in the **TableDemo** output listing in the Appendix.

Third, three **RelationVector** objects are created: (1) an airport relating as source to some routes, (2) some routes relating to their destination airport and (3) some routes relating to the airline serving them. Then the airport of the first relation's from-index's table row and the airline of the third relation's to-index's table row are printed to make sure everything is working.

Fourth, a tuple is created from the three relation vectors and printing of the same two rows above are made to corroborate that that works from the tuple.

Fifth, a **QueryPlan** object is created using those three relations in the same order as the parameter pack of types. Then, a synthetic **JoinedRow** is constructed (scaffolding) and printed to make sure joined row printing works.

Sixth, a nested loop join is performed to print out all the valid air routes (having valid airlines and destination airports) with Fresno Yosemite Airport as the source airport. There are 20 routes in the output, each with four rows printed: source airport, route, destination airport and airline. True to the intent: three relation vectors soft-define a joined row with four table row's K-values in it.

The first two K-values in the joined row element are soft-defined as (1) the from-index table row and (2) the to-index table row of the first relation vector and the second two are (3) the to-index table rows from the second and (4) the third relation vector, and that would be true for any succeeding relation vector. That definition isn't written down anywhere in code other than comments. It arises from the compile time generic programming structure of constants generated by the compiler. In other words, C++11 magic constants arise from generic programming recursion.

It is thought that soft-definition of type structure and member function by generic compile time programming recursive processing of other template class types is a new and somewhat open-ended method of deriving types, only limited by the imagination.

Finally, a new relation is created from an airport's country to an airport's country (perfectly legal.) What is not legal is to insert it into a QueryPlan before an airport table's index is included and then try to generate an object from that plan. The final test produces a relational linkage exception that is readable on the end to terminate the program.

Before throwing that exception, the time it took to execute the entire TableDemo program is listed:

1. Allocating and loading three tables from CSV files on disk = 29,748 microseconds.
2. Allocating, copying the keys and sorting ten indexes on those three tables = 91,222 microseconds.
3. Executing multiple point and range query lambdas on the indexes, as a test = 9.428 microseconds.
4. Creating three relation vectors, each containing a from-index and a to-index = 0.909 microseconds.
5. Creating a tuple from those three relation vectors to store the query objects = 0.448 microseconds.
6. Creating and optimizing a database join query plan for those three relation vectors = 2.561 microseconds.
7. Creating a joined row output array and doing a query plan nested loop join into it = 42.813 microseconds

All of that building, launching, query planning and generation, and nested loop join querying of a relational database from text editor files took 121,027 microseconds: less than 1/8 of a second.

If a consumer was using mmap to access this relational database as a previously produced slab, this execution would be much faster, especially on a memory centric Gen-Z system, or at least a 3D XPoint memory. With mmap it would take an estimated 1/100th second to bring up this slab database from scratch with one mmap call and execute the various queries in the `main()` of TableDemo. And that would be true of the startup loading of the mmapped database from scratch for the **JoinDemo** and **BigJoinDemo** programs as well.

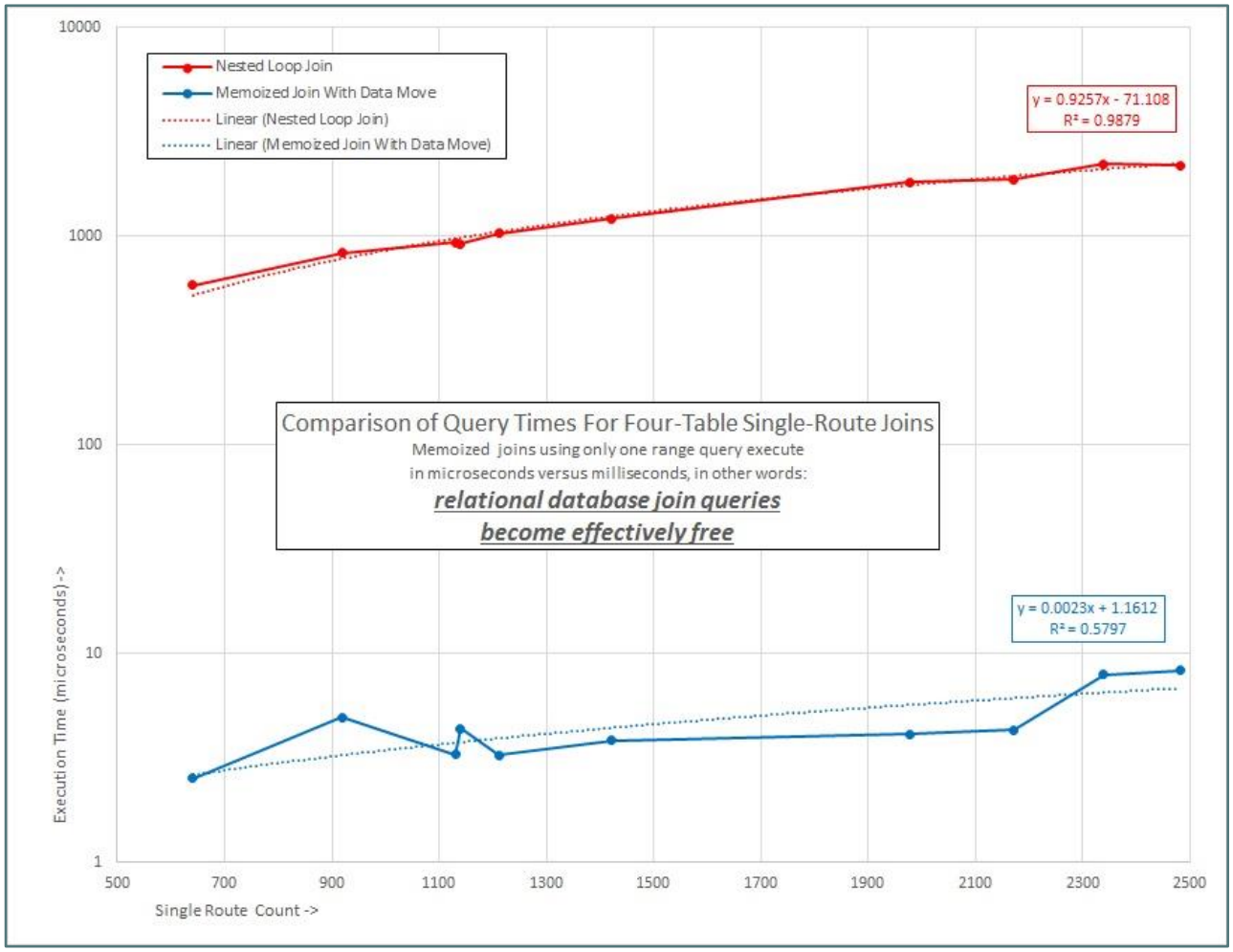
9. JoinDemo program

JoinDemo executes query logic on the scale of the entire size of the database. The relations are slightly different, but it uses the identical schema presented in the picture above, simply going at those 65,612 valid air routes from the access path of airlines instead of airports. The query for **JoinDemo** identifies and counts the valid routes for the top ten air carriers in the world. The output for **JoinDemo** is in the Appendix after **TableDemo**.

The code is very similar to TableDemo up to the point of completing the QueryPlan type and object. Three modified **RelationVector** objects are created and incorporated into a different **QueryPlan**: (1) an airline flying some routes, (2) some routes relating to their source airport and (3) some routes relating to their destination airport. Whereas **TableDemo** viewed the total set of

valid air routes from the aspect of airport, **JoinDemo** views the identical total set from the aspect of airline.

First, a lambda is created to execute a standard nested loop join query on the database yielding all the air routes for a specific airline, and then printing the performance statistics and the count of that airline's air routes. That standard join lambda is executed for the top ten airlines (by passengers carried) in the world.



Two things are apparent from the data: (1) Ryanair/Ireland, the number five in passengers carried has more routes than any other, presumably with shorter hops and fewer passengers per flight and (2) there is a correlation between (a) the number of identified valid air routes, (b) the number of range queries executed in the nested loop join, and (c) the execution time.

Roughly, that comparison reduces to one valid air route, per two range queries, per microsecond. The time complexity, or scale up in the common parlance, for nested loop joins on this relational database system is thus measured as linear time, with a record tiny scale factor (IMHO.)

Second, a memoized join is created recording all of the valid routes for all airlines in the very compact form of 16 bytes per joined row. This takes 69,235 microseconds for all 65,612 valid air routes and once again: roughly one valid air route, per two range queries, per microsecond for that nested loop join to execute in linear time. The cost of storing that memoized join is 1,049,792 bytes (~1MB.) FYI, any indexes (using the **JoinedIndex** class in the future) built on that memoized join for doing relational database "select" operations, for any column of any participating table, would cost 262,448 bytes (~¼MB) to store. Each corresponding index of **JoinedIndex** objects would take roughly 9 ms (linear time) to build on-the-fly, according to the sort data, but of course in an mmapped slab database that cost would only affect the producer, not the consumer.

Once again, the ordering of the future **JoinedIndex** object by an external index will require no storage of an index string in the joined index element, even to allow range queries on the joined row array: hence, **JoinedIndex** objects will not contain strings, only a single K-value and the type inference to an index element or table row containing the string. This will be also be true for combination key indexes on the tables and on the joined row arrays containing the memoized query data. The type information is in the template directing the parameter pack recursion to soft-define these relationships: once data is stored somewhere it is accessible in any combination of inferred relationships.

Third, a lambda is created to execute a memoized join query yielding the same effect as the standard joins above in the first step, including creating an array of joined row output containing the air routes for a specific airline, and then printing the performance statistics and the count of that airline's air routes. That memoized join lambda is executed for the top ten airlines (by passengers carried) in the world.

The results of the ten memoized joins are identical to the standard joins, except for the performance numbers. Of course, there is only a single range query for every memoized join query, because a memoized join query consists of a range query on the memoized join output array for the entire database. There is a rough correlation between (a) the number of identified valid air routes and (b) the execution time: that reduces to roughly 2-4 nanoseconds per valid air route reported, which is a speedup of two orders of magnitude for memoized join queries over standard nested loop join queries that require a couple of range queries per output joined row by the time you reach the bottom of the join recursion. This kind of relational database query performance at the nanosecond level for each single or point query result is unprecedented (IMHO.)

Memoized joins do effectively require that you "compile" the database along with the database code, but you only need to do that one time to create an mmapped slab that can be shared countless times at memory access speeds in the proper environment (Gen-Z, 3D XPoint.)

10. BigJoinDemo program

Since there are only 65,612 valid air routes in this database, simple queries on valid routes will not allow bigger results to test the performance at scale. However, what if the question is asked of the database, "How many double routes would there be if at the end of any flight on one of those 65,612 valid routes, the passenger hopped on any plane from any airline leaving that destination airport to go anywhere you could fly?"

Those could called "double routes" where the destination airport of that route was linked as the source airport of any other route for any airline. It's perhaps a stretch, but this could be an interesting query for disease transmission, tracking criminal activity or missing persons, etc.

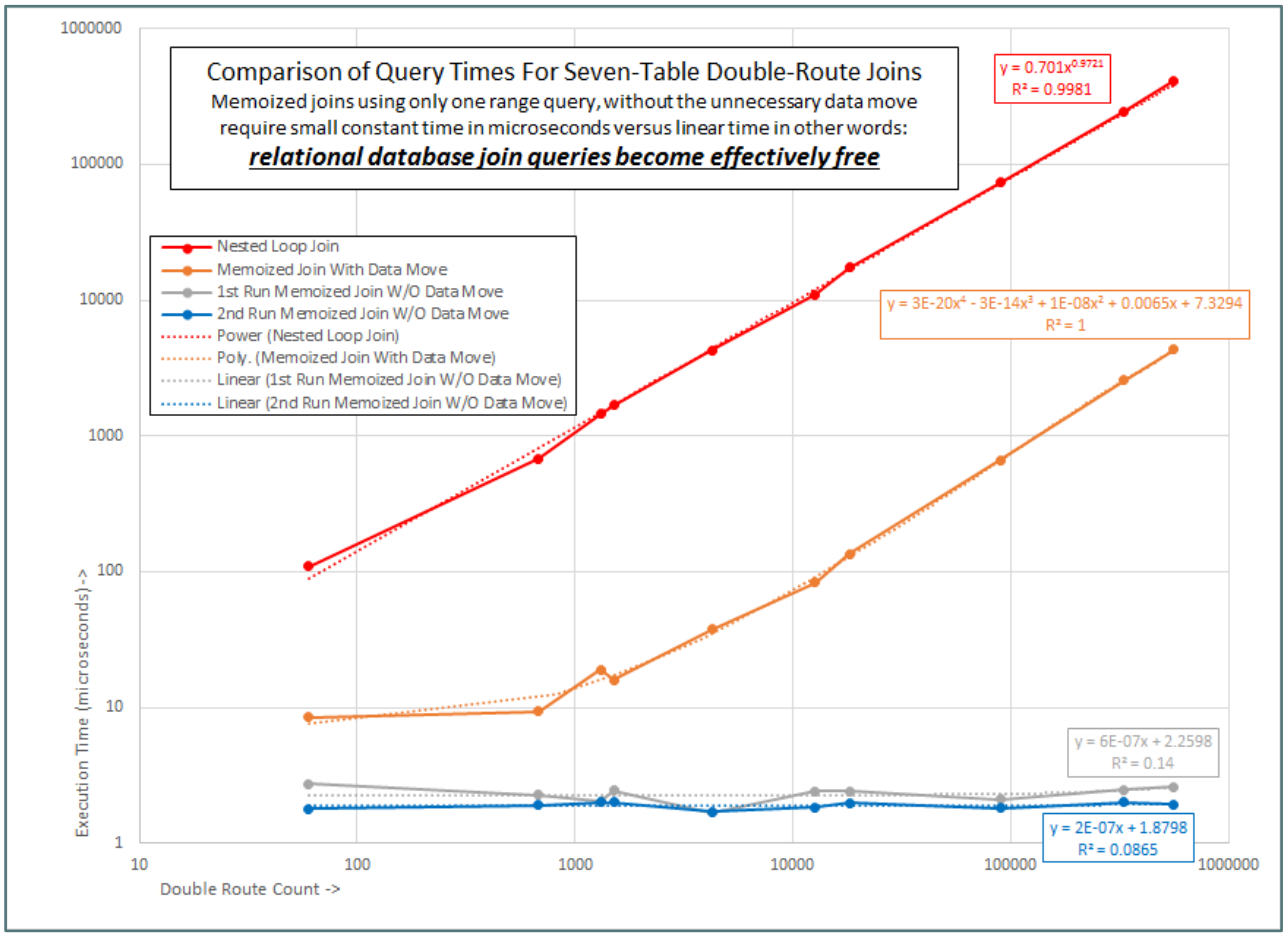
Double route **QueryPlan** class objects would have six **RelationVector** objects as template input types, of which one would be the same: the connection between a route table destination airport and the airport table row would occur twice. Hence five relation vectors are created for this double route join query that has six relations: (1) an airline flying some first routes, (2) some first routes relating to their first source airport, (3) some first routes relating to their first destination airport, (4) some first destination airports relating to second route source airports, (5: identical relation vector as 3, but different linkage) some second routes relating to their second destination airport and (6: inverse of 1) some second route airlines relating to an airline.

Since the top-down query plan construction in `constexpr void initJoin()` mimics how nested loop joins are done by hand, the query optimizer gets this right for this query and probably for any bill-of-materials explosion on inventory tables and similar queries that conform to the nested loop join model. This will not automatically work for more exotic queries, and future refactoring the current demo code base will naturally enhance query optimization to handle more and more exotic cases. However, this should never be at the cost of raw performance and scale.

BigJoinDemo has different subjects of inquiry from **JoinDemo**, which looked at the top ten passenger carriers. This demo looks at a relatively even trend of air carriers from the tiny air carriers like Alaska Central Express that only fly a few routes to the top airlines in the business. In addition to the two types of queries run in JoinDemo: the full nested loop join and the full memoized join producing the same array as the nested loop join for fast memoized queries, a third query only accessing the range on the memoized joined row array is done. This is because that memoized array, which is ordered by the from-index of airline provides the same sequential access to ordered result data as do the output joined row arrays from the first two queries, without moving the array data to another location. When the result data is small, this does not matter, but in the case of American Airlines there are 562,148 double routes in a little more than 15MB of data being

moved. Since that data move only gains you a difference in location and not in access speed, it is truly unnecessary and is quite costly for bigger result sets.

Interestingly, the second memoized join query that ends with moving 15MB of data wipes out cache, so the third memoized join query without the data move (the 1st run gray line on the chart on the title page of this document) shows the worst case performance after cache has been polluted. Thus, that test is run twice, to show the best case performance (the 2nd run blue line on the bottom) both showing constant time in relation to the number of output joined rows (double routes.)



The third double route query without the move is the fastest (in both the 1st and 2nd runs,) with the identical output row count to the prior two slower queries. The performance results are on the chart: a constant time of a couple of microseconds for joins, of any number of tables (C++ compiler limited,) for tables of any size (memory and operating system limited,) with mmapped relational database slabs sharable across nodes in a datacenter at the hardware limit of data access.

It took 11.011 seconds to do the full nested loop join to create the memoized joined row array for all 10,817,108 possible double routes in the database, and to allow those memoized joins to take on the average 1.9 microseconds (best case) to 2.23 (worst case) microseconds and make relational database joins effectively free of cost. Note that the correlation is still one microsecond per

resulting output memoized join array element during the nested loop join "compilation" of the database even at that scale. The time spent "compiling" the relational database into memoized joins by producers will dwarf the time spent compiling the C++ code for them, but it is time well spent and energy and cooling costs saved, if the database is used over its lifetime to produce more than 250 to 500 rows of query output.

Triple routes would most likely number around 2 billion or more output rows, and that could further make the scalability case, but the picture above makes the case sufficiently, since each query on the triple route memoized join result would still only cost one range query [at an average of $2(\log_2(n)-1)$), where n is $2x10^9$ array, instead of a $2x10^6$ array] to access as well. Since $\log_2(10^9)$ is only ~50% larger than $\log_2(10^6)$, and of course there is only one range query in both cases, an extra microsecond increase for every 3 orders of magnitude in total queryable scale is small.

Hence, the memoized join without the data move is arbitrarily scalable, since increasing the size of the memoized joined row array by 3 orders of magnitude only increases the join time by an estimated half microsecond (best case) or one microsecond (worst case.) And since increasing the width of the joined row element (more tables in the join) does not increase the number of hops in the binary search, adding more tables to the join does not increase join query access time at all.

However that is, after the output result arrays are created, the base processing cost (loading and storing to move it) for any single joined row resulting array element by indexing that array slab is estimated at 2-4 nanoseconds.

11. What's missing in the current code base (To-Do-List)

- Other `char` types support: `char16_t`, `char32_t`, and `wchar_t` will need to be refactored into the code, maybe increasing the code size by a factor of two. Much `char` functionality is missing for those other types and it can be quite diverse. This could fall under the heading of national language support, which from experience has been a challenge for SQL.
- Other string sort orders: The sorting done in the current code base is not really humanistic, but could be considered internal and human readable displays could be sorted on output. This is a refactoring issue and could fall under the heading of national language support as well.
- Fundamental types support: This needs to be refactored in, although the sorting and comparing speeds of these types is very fast compared to strings. These binary

representations are standard across C++11 for x86_64-based Linux systems and perhaps for other x86_64-based operating systems, and also for ARM-based systems as well:

  - Integer types (signed)
    - `signed char`
    - `signed short int`
    - `signed long int`
    - `signed long long int`
  - Integer types (signed)
    - `unsigned char`
    - `unsigned short int`
    - `unsigned int`
    - `unsigned long int`
    - `unsigned long long int`
  - Floating-point types
    - `float`
    - `double`
    - `long double`
  - Boolean type
    - `bool`

- User-defined types: As long as those types conform to base + offset addressing (no virtual addressed pointers contained in the type) this should be another small matter of software. Otherwise it breaks the paradigm and cannot be allowed. Basically, the storage format and the in-memory format must be identical, although external virtual pointers are allowable in the runtime code, otherwise these slab-based objects would not compute.
- Blobs: same as the user-defined types above.
- Advanced .csv file support (commented strings with full escape character support) should support any .csv files produced from commercial databases and systems like R, Mathematica™, etc.
- Usage of mmap: although not in the current demo code base, this should not be a stretch for atomic single writer/multiple reader support on Gen-Z and 3D XPoint. For distributed file systems, more work may need to be done in support.
- Other kinds of joins, more complex and exotic query plans, alias table names, etc.: This will be an ongoing evolution beyond schemas conforming naturally to the basic nested loop join model that the demo code base currently supports.
- Null values: During a two day course in relational theory at Compucon in the early 1990s by C.J. Date, who along with E.F. Codd invented relational database at IBM, Mr. Date was very clear in his condemnation of SQL nulls as solving a small problem that could be solved otherwise while creating enormous problems that have plagued the language since. So, this feature would be resisted, unless Mr. Date has since changed his views, since

exception handling in the inner loop will slow things down considerably. From looking at the Third Manifesto's latest paper, and here, both Date and Darwen continue to back away from nulls. They have found that once customers get used to the absence of nulls, they are OK with it.

- **RowString** high space complexity (wasted space): Due to all strings in the table array having the longest string fixed length: this can be changed by not using a **Direct**, but instead using a block of packed string body data, and having a 64-bit K-value that points directly into the packed data for the **RowString** body leading byte. Call the two classes **PackedRowString** and **PackedIndexString**, with better space complexity (overall size) and worse time complexity (overall cost of execution). It is possible these classes can be coalesced into one **RowString** class and one **IndexString** class by analyzing the template parameters of array size in number of elements or maximum string length.
- Additional "select" predicates: once **JoinedIndex** and combination key indexes were added to **JoinedRow**, you would still have the problem of how to successively reduce the output set of join row elements by additional "select" operations: this could be done by a successive selection technique. (1) Copy the result output array from the range query (first "select" predicate) onto a memoized join array, (2) build an index on-the-fly for the second "select" column or combination key (should be quick even for large result sets) and (3) then range query for the second "select" predicate, and repeat steps 1-3 for the third "select" predicate, etc., until converging on the final result after the Nth "select" predicate is processed, or until the empty set occurs.
- Aggregate functions: Such as `min()`, `max()`, `count()`, `avg()`, `sum()`, etc., these will require some analysis, probably a simple matter of software.
- Binary execution security in the middleware and user domain: Using Windows 10 Sandbox, and Linux Secure Computing Mode - (seccomp) expeditiously, this becomes a deployment problem that is solvable. As of now, there appears no need for any special seccomp filtering, the relational database only reads .csv files and writes to the console. After refactoring in more features binary security needs to be addressed again as a continuing issue.
- Privacy of data: Compression formats exist that support encryption of private data for special function analysis. This is an unknown development impact for middleware and user domains, but probably not necessary for datacenter usage (inside the circle of trusted interfaces.)
- Multi-processing: Processes can be a crash-consistent unit of failure, threads cannot be a crash-consistent unit of failure, and threads cannot span processes or computer nodes due to virtual addressing limitations and subsequent composability problems at scale including

cascading failure or inconsistency resulting from one thread failure. Thus memory sharing between processes, across nodes and different operating systems, must be through mmap without virtual addressing and necessarily by base + offset addressing at scale. This is a debate that can only end in one result, read the **Futures.c** section below to see how that is resolved.

- Enormous scale: Read the **Futures.c** section below to see how that is resolved.
- Mutability of the Immutable: The current demo relational database system performance and scale requires the database to be immutable. Read the **Futures.a** and **Futures.b** sections below to see how change could be managed.

12. Futures

a. Israeli Hybrid BASE + ACID Database Systems

In early 1983, a team from the CACI™ London Office managed by George Mahelona and led by Joanne Wheeler, flew in to Tel Aviv, Israel to work for Lieutenant Colonel Elie Ben-Dan, the IT visionary of the Israeli Air Force to develop OLTP software on Tandem Nonstop computers for the MIRS/MADA project automating the maintenance logistics system for the F15/F16 squadrons on the 11 air bases spread throughout Israel. The local development team was from Control Data Israel, and were more used to the fail-stop mainframe systems, than the Tandem message-system-based fault-tolerant clusters distributing transactions across the slow buses and networks of the time.

Screen Verification Response Time Problems from Slow Disks

Joanne was a dynamo originally from the elite team of developers sent out by Tandem to evangelize and spread the NonStop faith: she quickly taught the developers to understand how clusters worked and explained the manuals written in English, not Hebrew, so that they would *not* get stuck reading one page for hours. Joanne got the database design squared away, backing off from fifth normal form to first or second to drastically reduce the number of programs and multiple maintenance, which involved programming nested loop joins by hand in COBOL servers against the Tandem NonStop Enscribe quasi-ISAM database (there was no SQL yet, only a relational report writer called Enform.)

However, there remained a serious problem with performance. The maintenance reports were entered from paper forms and contained dozens of jet mechanic-memorized mnemonic codes stored on the database in about 40 smallish code tables. The data entry screens thus had dozens of

fields on a screen that had to be verified, and after hitting a function key, it would take 100 seconds or so on a NonStop 1 to get the cursor back reflecting the mistaken code entries. Not very satisfying!

Speedup of Screen Verification by Three Orders of Magnitude

How to speed this up? After reading the manuals on the TP Monitor (Pathway) it was noted that there was a facility called User Conversion Routines that could be bound into the TP Monitor image as a library written in the system language called TAL. TAL was a complete rip-off of the SPL language and code from the HP 3000: same code, same instruction op codes, which the Tandem folks just grabbed all the technology from when they left HP (those were the days of boardroom-VC handshake deals.) It was also learned that extended segments called QSegs could be shared across processes in the same cluster node at the same virtual addresses above the 16-bit address limit of 64K bytes.

Then the design was forged to have a fault tolerant COBOL process pair launch and maintain daemon processes in each of the 2 to 16 cluster nodes (called "NonStop cpus") in a cluster (called a "NonStop node",) with each daemon reading in the 40 code tables into QSegs using the NonStop serial I/O library (speedy) and then projecting the appropriate columns in fixed length, and quick sorting the table by the verification columns so that binary search could verify the fields on the screen of the TP Monitor or flash their text description, where those screen-fields had coded numbers numerically identifying the table in the appropriate QSeg, and then looking up the screen-field text for verification and reversing or blinking the field color when it wasn't there.

The daemons would then monitor the database table's file system metadata every minute for changes in the modification timestamp and if modified, then release the obsolete QSeg for that table and make a fresh one. The TP Monitors would notice when the modification timestamp in the metadata of a QSeg that hadn't changed in 3 minutes and release it and go get a new one from the daemon.

When the developers tested this code they loaded a screen with code values and hit the function key and thought it was broken. Then one developer suggested adding some wrong codes and after hitting the function key the screen flashed the errors in the blink of an eye. Thus was born the world's very first BASE (eventual consistency) + ACID (fault tolerant and crash consistent) Hybrid database system. (That can be said with assurance, because none of the databases on minis or mainframes at the time were fault tolerant: Stratus had just come out in 1982 and had just a few accounts in development.)

Almost twenty years later, upon meeting an Israeli developer at the Tandem ITUG yearly conference, it was found that they were still maintaining this code on the system. Three orders of magnitude speedup at the front end is addicting to busy humans!

Hybrid Snapshot Eventual Consistency

The current relational database demo code base could also be run Israeli-style, with any ACID database that can produce .csv snapshot files, or any groups of diverse databases from diverse manufacturers for that matter, and then that could support queries that execute at any scale or number of tables, with complex joins that are constant time in microseconds, at the middleware level, or close to or located at the front end where the user sits.

The process of automating the collection of data from the various databases could be managed by using a log reading system like Gravic™ from the Holenstein brothers and Bill Highleyman. Gravic™ reads logs from the big system manufacturers and can maintain consistent replicas in a system of choice, which could then be used to periodically produce new relational database slabs, using the current demo code base, replacing the old generation slabs on the Gen-Z Librarian File System or a distributed file system for mmap consumption.

Then, as in Israel, the obsolete versions would get flagged, or not flagged and age out, as the case may be for switchover. There might be a minimal open source replacement for, or version of Gravic™ supporting the popular database systems, if not, someone should probably produce one maintaining .csv files. It should trigger, or send a signal, or do a database version aware multicast, when a newly-defined set of .csv files have snapshot consistency (if they are snapshot-consistency-aware,) or the host application itself could materialize beacon values in the database to allow snapshot-consistent-awareness in log readers. Just an open source thought.

b. Native ACID Transactions Mutating the Immutable: Viewed from the Aspect of WAL

The immutable relational database demonstrated in the current code base may seem foreign in comparison to the traditional SQL ACID transaction write-ahead-log (WAL) systems that users are more accustomed to. However, when viewed through the lens of how those systems have gotten decent performance over the decades from slow disks avoiding glacial random writes, the immutable relational database system in the current demo code base might look a little more familiar. WAL systems stream serial writes containing the database changes to the log along with the commit records for the transactions changing those databases, before ever allowing a single

page with committed values to get flushed onto the database disks. Kind of like temporal immutability, looking like what modern evolutionary theory calls punctuated equilibrium?

Then to limit the pain of crash recovery, periodically the up-to-date pages kept in memory (and safely stored on the log) are flushed out to the database in as few serial writes as possible called a database checkpoint (even on SSDs and memory systems, serial writes outperform random writes by a factor of up to dozens of times.) Kind of like making a new generational archive?

On all of the commercial databases, the version of the database on the actual database disks is never consistent. That is, unless you perform what is called a consistent system shutdown (which rarely happens) the disk version perpetually contains broken b-trees and sometimes worse. Most hot-spot data written to the database will never see the database disk, staying in the memory cache and never being evicted. This is why people say that "the log is the database" and that the database disks are merely an "inconsistent cache" ("people" include Jim Gray, Pat Helland, Shel Finkelstein, Jimbo Lyon, etc.)

Crash recovery is the process of putting the lost in-memory part of the database back into memory from the log. Crash recovery should read the on-disk database as little as possible, and really try very hard not to write to it with what would mostly be random writes (slow,) at a critical time. Kind of like the ingested changes on an eventual consistency database are always going somewhere else than the archive copy of the database?

So, like anything magical, database performance is really misdirection, a shell game for rubes. That implies that doing a database query is really performing an artful union of the smaller and active in-memory piece with the vast inconsistent piece on disk, like showing the stunned rube which shell the pea is really under and then taking his money. Modern databases only present the Darwinian view of the smooth evolution of the database, when what's really going on is the Eldridge/Gould reality of punctuated equilibrium, where the punctuated interruptions are really preparations for chaotic failure. Remember that word: "union."

LazyBase/Metabox at HP/HPE Labs

Back at the beginning of this decade, a project at HP Labs called LazyBase was experimenting with eventual consistency to get the maximum possible data rates for both ingest and queries. The project team of Kimberley Keeton, Charles "Brad" Morrey III, and Craig Soules was led by soon-to-be HP Fellow Alistair Veitch, and their work was fueled by an endless stream of PhD-student interns. LazyBase became Metabox, and based on the DataSeries software performance that saturated the disk channel by Alistair's other HP Labs team of Eric Anderson and Joe Tucek,

Metabox became the tech-transferred project software for allowing the HP and later HPE StoreAll distributed file system (DFS) of an archiving system (the DFS was Ibrix at that time) to have a constant time of two seconds or less to complete complex queries with eventual "prefix consistency" for the file system metadata on distributed file systems of literally any size. "Arbitrarily scalable," Brad called it, using the mathematical definition for arbitrary.

Metabox materialized immutable generational meta-databases in DataSeries extents that were compressed and accessible by key range allowing the user to go quickly to the one page of anything in the archive that was desired. It was an amazing accomplishment on the rotational storage systems of the time, a triumph of serial access over scale.

Eventually, HP/HPE tired of Ibrix issues and the rug was pulled out from under Metabox development. Before that happened, Brad and Kim developed some ideas about how to give Metabox ACID transactional up-to-date consistent metadata queries while retaining the massive ingest speeds and constant time for those queries. The answer, in a word, was "union."

LazyBase/Metabox (see paper, Figure 1) had two faces: an immutable archive of enormous size and a pipeline of ingested changes in stages of incorporation stored in Linux file system directories. Periodically, a new archive would be generated. Fault tolerance would be supported by keeping the old archive around and writing ingest to two sets of pipeline directories, old and new. The key to up-to-date (not eventual) consistency was to make the pipeline capable of supporting queries and to preferentially union the pipeline query results over and on top of the archive query results (a relational application of the Thomas Write Rule.)

The hope was, that since the pipeline is periodically cut back to remain small, those up-to-date consistent query results would be effectively constant time, and since the archive queries were already constant time at any scale, the union of the two might be constant time as well, ignoring the complexity of that relational union operation. Then super-fast eventual "prefix consistent" queries could be augmented by slower, but still effectively constant time ACID transactional up-to-date consistent queries. Replication for disaster tolerance would be handled by keeping multiple generations of the entire complex around and streaming the same ingested changes to each of them in parallel as quasi-mirrors. That was as far as they got, however, before the denouement of Metabox.

Both LazyBase and Metabox at an earlier point, used ID remapping internally (see Figure 1 again, page 4), their SCUs were like transactions (IDs were similar to the K-values used in the current code base, but only immutable until mass remapping) to link up ingested and stored values in the pipeline using sorting and merging. ID remapping was abandoned later for Metabox, since at scale

it turned out that it was faster to sort the keys themselves and abandon the intervening IDs. Before that happened, Bloom Filters were looked at as a method of speeding scaled distributed access to the IDs.

Welding a Bloom Filter to a Log Tail Database

Bloom Filters (BF) are interesting, they give excellent scalability for checking the presence of an identifier, like a key value. They give adjustable (by BF size) false positive matches, but zero false negatives. If a key is not in the BF, it's really not there, whereas if it is in the BF: that ambiguously hints that a more costly search must be performed to get what you are looking for, which some of the time is a phantom. That is perfect for ingest union queries, because that means that whenever your identifier is missing from the BF, you can always use the immutable database query results that you must run anyway for that identifier, with total consistency.

Then the initial design for the ingest side would atomically weld together a scalable log tail database and a BF, in that causal order, such that the database was logged in a manner that allowed queries of what's in the tail end of it and that state was reflected immediately after and kept atomically consistent within the BF. The transaction and logging part needs to scale for maximum ingest, with log partitions and a log root for commit in a fault tolerant cluster, supporting distributed three-phase commit across many log roots for scale out, with distributed transactions and the VSN-style log-streaming that NonStop employs for massive scale up and out. More on that in section 12.c.

The magical part would be how an ingested change is transmitted, to easily union with the nested loop joins and the memoized joins. This requires an inversion of the join logic. The base + offset coded slab BF would have to support modified bit lookup by [table (`int`), row (K-value), column (`int`)] triplets relative to the current slab generation being queried. During an up-to-date (not eventual) consistency nested loop join operation each column value being used would check its bit for modification and if the bit is clear, carry on. If the modified bit is set, the deeper look would get the new column value, or find that row was deleted, or find the false positive and do the appropriate thing.

Inserted row modifications would require a different approach affecting walking the array indexes and range queries. For inserts, you would want a separate BF for insertion bit lookup by [table (`int`), row (K-value)] doublets, which if set would lead to an ordered list of inserted rows at the insertion point. After every new archive generation, the log tail databases and the BFs would get zeroed out atomically, leaving the log itself for log reading purposes and logical-rollforward-style-recovery from software bugs and bad data entry.

Memoized join objects would require some relational calculus to be inverted on those operations: sort of the like the partial differentiation opposite of multiple integration in calculus. A nice research project for someone, it's just more mathematics. Until that is worked out, up-to-date queries would require actual nested loop joins and could not use memoized joins. There has been recent work on scaling out replicas of BF images, which could allow remote access to the BFs outside of the original mmap datacenter.

c. Parallelization: Scaling Up **and** Out

Massive scalability requires the highly reliable kind of fault tolerance that is optional for small scale (toy systems.)

Massive Scalability, Fault Tolerance, Availability and Unit of Failure

Massive scalability requires high availability, because low availability systems have longer and longer periods of outage, as stateful systems with dependencies scale up and out (e.g., as opposed to stateless silo redundant copies of websites with no interdependencies on highly reliable networks.) This is less a problem of finding and removing the causes of failure than it is of shrinking the repair time of systems, due to the availability algorithm:

$$\text{Availability} = \frac{\text{MeanTimeBeforeFailure} \equiv \text{Uptime}}{(\text{MeanTimeBeforeFailure} \equiv \text{Uptime} + \text{MeanTimeToRepair} \equiv \text{Downtime})}$$

In the limit where the MeanTimeToRepair≡Downtime shrinks to zero, Availability quickly approaches one, independent of the causes of those failures. Hence, shrinking repair time is an urgent necessity for massive scalability and hunting down and preventing increasingly problematical and thorny sources of failure that come and go as the architecture evolves is like mining for low-grade ore at some point: thankless until repeated failures become an irritant.

For fault tolerant and highly available massive scalability, multi-processing is profoundly stronger and more reliable than multi-threading.

This is because processes updating shared memory can be and are, in practice, discrete units of failure (e.g., Tandem NonStop process pairs sharing virtual memory mapping for full stack checkpoints across a fabric are a proof point, but are not a useful model, whereas Atlas and Ken prove this point in a useful way,) but threads updating shared memory cannot in practice be discrete units of failure within a surviving process. The bigger the thread count of a multi-threaded

program taking over a large multi-core node, the larger the crash and the recovery outage when a single thread fails. Also, larger and more elaborate crashes incur more bugs in recovery.

Data sharing in threads is done in the cache hierarchy hardware via coherent memory that snoops for changed cache lines across cores, invalidates cache lines in all cores on modification and makes room for reads and modifications in local cache by cache line eviction. Threading with coherent virtual addressing will never be distributed across nodes either reliably, or at scale, because of the non-composability of aligning all of those virtual address spaces, so the hack of multi-threading across nodes that might solve this problem is never going to work at scale. Also, changing the thread model to look like processes to allow them to be discrete units of failure would be redundant, because we already have processes.

Why can't the current thread model be made a discrete unit of failure? The failure of a memory updating thread inside a process cannot in practice allow a reliable program to continue executing: thread failure escalates to the unit of failure of the process for crash consistency of state. This will not change, because of the complexity of the cache hierarchy hardware state in the cores and the uncore of the processor in support of multi-threaded consistency, and also because the effects of that hardware state cannot be objectified on process or computing node failure and recovery. The only state that can be externally objectified on failure and recovery is the state of the DRAM memory on process crash and the nonvolatile state of the system (non-volatile memory and media) when the computing node that contains the process crashes.

Parallelization of Non-Coherent Memory Sharing With Base + Offset Addressing Across Process and Node Boundaries, Using Single Writer Processes and mmap

The combination of mmap and base + offset addressing treats storage memory hardware as non-coherent, not relying upon the cache hierarchy for anything other than keeping track of the immutable virtual address of the base in that process accessing the mmapped media. In practice, since common virtual memory addressing cannot be composably and reliably stretched across a large number of processes in the same computing node, then it is unimaginable that paradigm could be made to function reliably across computer nodes that can individually fail, different operating systems releases, and finally different kinds of processors like GPUs and different operating systems (Linux, Windows, AOS, etc.)

Attachments to methods that do not massively scale while retaining composability must be abandoned at some point: multi-threading and virtual memory simply must effectively be discarded

beyond systems larger than one node, to scale memory access to allow efficient data processing on the massive distributed memory systems emerging now.

Upon acceptance of that inevitability, literally everything done in the current demo code base can be multi-processed and distributed in parallel across nodes in the mmap-space of Gen-Z central memory or (with greater difficulty) a DFS supporting on-memory-page-fault-demand-based mmap (as opposed to the Java or Python mmap call-based implementations, IMHO), and even across different operating systems, and still remain shared and consistent.

Table fragments can be (1) loaded in parallel (using carefully partitioned K-values from precise row counts) and (2) joined by a simple union in K-value order. (֍ On Gen-Z shared memory systems, the final unions can be done in one massively parallel step onto one multiply-mmapped object, but that can't be made to work correctly on DFS lacking the memory fabric atomics, it will require a single-threaded step to do the unions, IMHO.)

After the table fragments are loaded in step #1 above, index fragments can be loaded alongside the table fragments and partial sorted in parallel. Then like the ordered merge of the table fragments of step #2 above (similar to the second half of a distributed merge sort,) an ordered merge of the index fragments can bring them into a single index, in parallel to the ordered table union (֍ for Gen-Z systems, ditto the above for these.)

After all of the tables and indexes required for a memoized join are complete, the full memoized joined row array can be built by partitioning the initial column indexing key for the entire or partial range required by the full memoized joined row array (in the current code base that would be done on the from-column joined row index from the first relation vector) and (A) doing the nested loop joins in parallel to make the memoized joined row array fragments, and then (B) joining those memoized joined row array fragments using a simple or ordered union in the preferred order into full memoized joined row arrays (֍ ditto Gen-Z for these.)

As sorting is for indexes, and not the tables being indexed, sorting is also for memoized joined row array indexes and not for the memoized joined row arrays being indexed (the current code base violates that rule, due to the lack of joined row indexes.)

After the memoized joined row array fragments are loaded in step #A above, memoized joined row index fragments can be loaded alongside the memoized joined row array fragments and partial sorted in parallel. Then like the ordered merge of the memoized joined array fragments of step #B above, an ordered merge can bring the memoized joined row array fragments into a single memoized joined index, in parallel to the ordered joined row array union (֍ ditto Gen-Z for these.)

For combination key table indexes or combination key joined row array indexes, these could not be done by fragments in parallel to the above (except for some special cases), but would have to wait for all of the above to finish, for their construction. However, after that delay for consistency, many hands make light work and these can also be done in parallel ranges, too (🌀 ditto Gen-Z for these.)

Slab Databases

A consistent set of tables, indexes, full memoized joined row arrays and full memoized joined row array indexes can be combined into a single slab. (🌀 On Gen-Z shared memory systems, the final slab union can be done in one massively parallel step onto one multiply-mmapped object if all the full union sizes are known, but that can't be made to work correctly on DFS lacking the memory fabric atomics, IMHO.)

That slab organization and functioning requires a metadata section at the front with a count of offsets to the bases of the objects in the slab and a list of those offsets. The first objects in that list could be the C++11 and later source code modules, and documentation and the make file for those modules. Then the first step is to compile and run the code with a standard script, which code then accesses the slab, and then the relational database system and its applications are up and running.

Meta-joins could be supported across slab databases from various sources, which then could be materialized in memoized meta-joined row arrays and memoized meta-joined row indexes on those. Those memoized meta-joined row arrays and indexes would need to be rebuilt when the underlying generational slab databases get rebuilt as new archive versions emerge. However, they are tiny in comparison and this should be fast: the memoized meta-joined row arrays have one 4-byte K-value per projected column (not all underlying columns would be necessary typically) and the indexes would only have an array element of one 4-byte K-value on that memoized meta-joined row array.

Meta-slab databases could be created by extending the stated approaches above, and perhaps aren't necessary, but who knows? It would be efficient in practice and make life easier for deploying large complex applications.

Physical Slab Database Construction Using An mmap Variant

When using typical mmap against an immutable chunk of media on Gen-Z or a DFS, where do the modifications to memory go upon page fault? The answer is to local DRAM pages attached to the computer processor executing the "store" instructions modifying memory at the specific memory address and causing the page fault. The first memory store operation against an immutable

mmapped memory page will cause a copy-on-write (COW) allocation of a new DRAM page in the attached processor, overlaying the virtual memory to physical address mapping of the unmodified page privately for the modifying user, and leaving in place what all other processes in the computing node and other nodes will see as unmodified. It is the act of copying without the maintenance of consistency across processes by the Linux kernel that makes COW efficient and for the most part, bug-free (only kernel guarantees to maintain.)

Modifying applications will then have their own private view of any database shared using mmap (not just the current code base.) Going back to the "shell-game" analogy used in the sixth paragraph of section 12.b, above, for a WAL database using standard mmap this would be a private DRAM physically-addressed page of the b-tree block in DRAM (at least) seen only by the modifying process, but for the current code base the changes would be to the modified pages of the log tail database with a bloom filter welded onto it, and never to the immutable archive pages, which would remain identical in all processes executing in all nodes across the network that have the archive mmapped, including the node of the modifying process who has a modified private view.

This establishes a standard mmap preference towards a single writer per slab, or per log-tail-BF-partition of a database log supporting a number of slabs, and likewise for WAL databases using standard mmap. It is also a problem for failure recovery, since local DRAM gets zeroed out on reboot after a machine check, and those private modified DRAM pages go into the bit bucket when a process dies.

Variations on the theme of mmap, for many purposes requiring sharing and process/node crash recovery from failure outside the standard usage, are widely found, as are many hacker exploits based on those variations. Because the current code base enshrines the shell game directly by never writing the immutable archive version of the database, the modifications to make a specialized mmap are fairly trivial and potentially easier to shield from hacker exploits. Whenever a valid page fault would occur and be serviced, that would only occur to the log-tail-BF-partition of a database log, whose mmapped media resource would be protected by Gen-Z Librarian File System or DFS security, and that page fault would cause the modifications from the original page to be sent to the log and immediately manifested and made visible in the attached bloom filter, for this partition of the merged log. Then both hybrid BASE + ACID systems of section 12.a above and the native ACID transaction systems of section 12.c above could maintain their corresponding eventual BASE query prefix consistency and their up-to-date query ACID transaction consistency.

Massive Scale Up: Treating Memory as Storage

Back in 1993 Jim Gray wrote *The Benchmark Handbook*, defining two terms, speedup and scaleup:

> *A system provides what is termed linear speedup if twice as much hardware can perform a given task in half the elapsed time. (p. 290)*
>
> *[There is a] difference between a speed up design in which a which a one-minute job is done in 15 seconds [4X the hardware], and a scaleup design in which a ten-times bigger job is done in the same time by a ten-times bigger system. (p.290)*

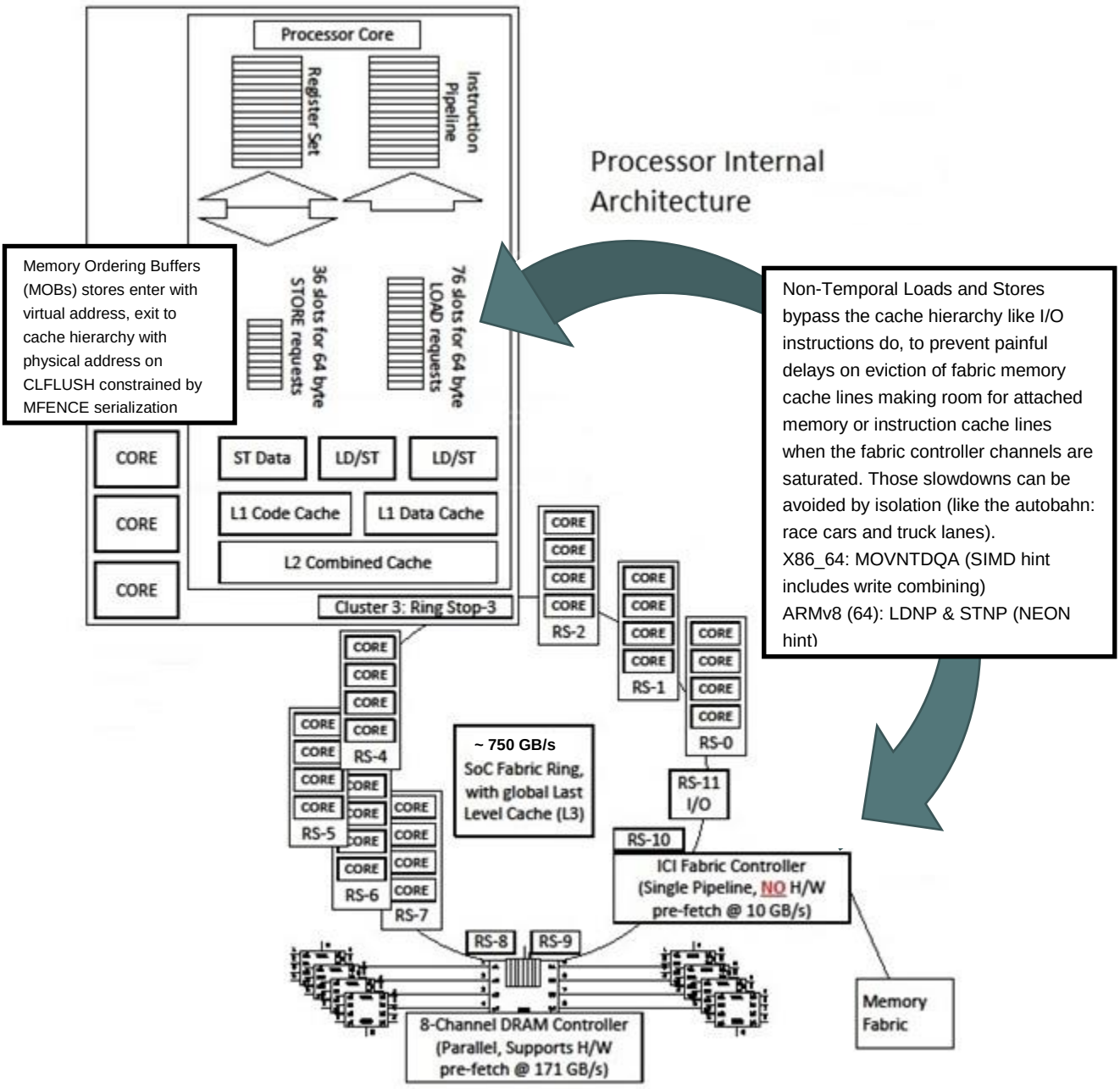
Then a good speedup curve (we now call this scale up) of old time divided by new time is linearly increasing. A good scaleup curve (we now call this scale out) of old time divided by new time was flat. A ridiculously good curve is where both are a constant time of a handful of microseconds independent of the load or how much hardware you have. That is what the current code base seems to offer for memoized queries.

With something as wonderful as enormous Gen-Z shared memory, there must be a catch. Indeed, that is NUMA (non-uniform memory access,) where some memory is locally attached and very low latency/high throughput for loads and stores to go through immediately like posted writes. Load operations get replied data, but there are no replies on store operations. If there is a problem not masked by ECC or Chipkill, you get a machine check and local attached memory is zeroed on reboot. However, loads and stores to fabric memory are not like loads and stores to attached and reboot-zeroed memory.

Since you have enormous shared memory on a memory-centric system, you do big sorts and block move processing operations by hauling heaps of memory in and out of the processor, like doing I/O-based processing. This is why I/O instructions bypass the cache to keep from rendering it useless, and the NUMA equivalent to those I/O instructions are “non-temporal” load and store instructions, so that shared cache (3rd level cache, last-level cache) is not polluted with cache lines that are slow to evict to make room when an instruction or local DRAM data needs to be cache line fetched in to execute or process.

Slow to evict can be very, very slow when the fabric controller, which is hung on the fabric ring like the I/O controllers and the DRAM memory controller, gets saturated. Fabric memory is farther away in spacetime, and there are fabric modes due to switches that attached memory controllers don’t experience. Attached memory controllers can have eight channels and abundant slots, so as to rarely saturate the queue. Fabric controllers *should* saturate if you are driving the system hard, to

do benchmarks. The limiting resource *should be saturated* if your software is working properly, at the hardware limit.



The problem is that however many channels and slots per channel you have, when the controller is saturated the queue depth is effectively oscillating slowly between zero and one, and all the cores doing memory fabric NUMA work are fighting for that one slot that comes free. If what is waiting

for that slot is the eviction of a cache line to allow an instruction to page in for a system process to run in a timely manner, the system grinds to a halt. The cure is in the diagram above.

The non-temporal load and store instructions are hints, which if you have the cheaper versions of the processors might just be ignored: their existence can be probed for in both X86_64 SIMD and ARMv8 (64-bit) NEON. The image is modeled (with many guesses on the microarchitecture) on the Vulcan ARMv8 processor of the recent past. The Memory-Ordering Buffers are driven and ordered by the CLFLUSH and MFENCE instructions on X86_64 and the equivalent instructions for ARMv8. The temporal load and store instructions are the MOVNTDQA (SIMD on X86_64) and LDNP and STNP (NEON on ARMv8/64-bit.)

Using those instructions, fences and cache line flushes correctly (a subtle and black art) on the right kind of metal, allows the code that is going against the fabric controller to potentially block on that operation and to bypass the cache hierarchy, leaving that for the fast local work, which hopefully is part of the algorithm. A smart distributed sorting system would then pull a chunk of rows into DRAM addresses from fabric memory, partial sort it there and move it back out to fabric memory from DRAM addresses, and doing appropriate choreography in parallel across a large number of cores in nodes. The term "DRAM addresses" makes the point that data moving through DRAM should never get flushed out beyond 2nd level cache from the core into shared 3rd level cache, certainly never to actual DRAM, hence sizing to 2nd level cache and careful allocation of loop variables would be a win. This is (1) treating fabric memory like I/O, (2) treating DRAM like 1st and 2nd level cache, (3) never caching any data from fabric memory, and (4) leaving the 3rd level cache for other system programming like a good citizen.

Massive Scale Out: Distributed Multi-processing

This involves some choreography, since the memory fabric is the limiting resource for distributed computing on Gen-Z. Your algorithms should use the memory fabric sparingly and do as much as possible against local DRAM, from and to shared memory on the local memory fabric switch. In a distributed sort, for instance, you start with your unsorted data on the local switch (near fabric) for each of the nodes and end up sorting it and distributing it widely (far fabric) to all the other nodes, finally merging locally on the fabric memory near each of the nodes. Common sense approaches will still require fancy choreography and a control channel, since at best skew is involved in sorting and that means a slow step somewhere needing help of some kind to reduce single-threaded performance. At worst, anything really big and distributed with a lot of moving parts involves the greater likelihood of partial failure and having to redrive a step, so that should be possible which means someone somewhere needs to pause for the redrive completion notification from everyone, probably multiple someone.

Those are the general issues for Gen-Z fabric distributed processing, now to the specific. For the current code base, the ingest rate for processing changes is all on the transaction system. Query utilization costs should be very light, because of memoized joined indexes. Compilation and database compilation costs will be greater, but still minor. It is the ingest costs that will dominate the system. The basic unit of a scalable transaction system is a clustered entity: the transaction service. The transaction service is clustered to maintain high availability, and it can be backed up by other clustered transaction service entities in a multi-cluster for 100% uptime, although there will be network partition events (disconnections) between the transaction service and distributed entities outside the data center (like users and other data centers.) However, that does not preclude total consistency when the network is not partitioned [PACELC, Daniel Abadi.] In the case of the BASE immutable archive of the current code base, network partition simply extends the wait for "eventual consistency."

For the ACID up-to-date part, which is a log tail database welded to a Bloom Filter (BF), scaling that up and out in a cluster involves multiple log partitions merged onto a root log that receives the transaction state records for group commit and abort, using a merged log algorithm. The log partitions (and the root log) track serialization by Log Sequence Number or LSN (Gray, Reuter: *Transaction Processing: Concepts and Techniques*, 9.1.2.) The LSN is only known ***after*** the log write is completed (too late for streaming log writes.) Merging streaming log writes to log partitions requires a Volume Sequence Number or VSN, issued by the transaction ingest resource manager (RM) logging, such as the mmap variant writing changes on page faults mentioned at the end of the previous section above. At transaction group commit time, the vast majority of transaction changes required to be received by the log have already been streamed to a log partition (WAL) and a check on the VSNs of the authoring RMs on their corresponding log partitions will guarantee that the writes are durable. That check is performed just before the root log commit write, to guarantee merged log serializability. If all these components are maintaining their data structures on mmapped sharable resources with base + offset addressing (with fabric atomics for access consistency) then that check just prior to the root log group commit write is a series of memory accesses, and not distributed messages as in Tandem NonStop. Distributed and or synchronous log replication can be done across transaction clusters with high availability via three phase commit as in Tandem NonStop. There are a long-winded series of slides and videos done a decade ago, explaining all that in great detail, here (the intro is ten years out of date, but starting from reliable message system fundamentals, distributed transactions, clustering, VSN log streaming, async and sync consistent replication, it's all good from hours 0.5 through 9.)

Fault Tolerant Process Singlets

Tandem NonStop does its fault tolerant computing with active backup processes called process pairs having the same identity, like Bosons, and with a cluster state consistency algorithm called Glupdate by Richard Carr, which does a service akin to Paxos, but has much better latency for failure takeover and only used on small NonStop clusters 4-16 nodes. With that they can get 5.5 to 6 nines of availability on their database, and have been measured in the British banking systems at up to 7 nines by using an async (called 1-safe in Gray, Reuter, 12.6.3) replication scheme called RDF, by Richard Carr and Malcolm Mosher. At an earlier time in NonStop, whole stack checkpointing using identical virtual memory mapping was popular, but that became unwieldy in practice for scalability performance reasons. Most system processes at NonStop now checkpoint their intent and then after takeover, idempotently retry all intended, but not checkpointed completions. Then duplicates in the log that survive are tolerated and duplicate requests are answered appropriately, if the results are consistent (e.g., no aborting then committing, or the reverse.)

With the ability to share data on a memory-centric system across nodes using mmap and base + offset addressing (on Gen-Z or DFS) as a single writer, the possibility of a different fault tolerance model using process singlets emerges. If a process has used shared resources that are immutable, like those relational database mmapped slabs suggested by the current code base, then only the process state itself must be maintained in a singlet mmapped slab. For process globals that are immutable or static, those can be written at startup into constant locations within the slab, but since "Update in place is the poison apple" – Jim Gray, atomicity for variables that come and go (call it the process's "desktop") after the process crashes or the process's node crashes, must be handled differently. A variation on an intern project at HPE Labs called TxHPC that was presented at NVMW would allow for crash consistency of process state, and also tolerance of shared memory media failure through erasure coding (using Ethan Miller's Gerasure, but it could be converted to use his latest and greatest Gferasure.) Basically, it keeps track of where its writing base + offset addressed state and those writes rotate in a predictable way such that after a crash, you were not halfway finished writing a variable onto non-volatile memory, or shared DRAM that does not get zeroed after machine check or node crash by panic or other reasons.

Then the only thing that is missing is a method for the timely notification of the death of the process and or the node containing that process that is registered with a distributed and reliable cluster service, which then restarts the fault tolerant process singlet with access to the mmapped slabs containing its prior global, static and "desktop" state, which is then recovered. Millisecond takeover has been pioneered by an HPE NonStop facility called "CPU Broadcast" by Mike Rice

and that is described here. Microsecond takeover on memory-centric shared memory systems is possible and that is described here.

Then the process singlet's operations that are moral equivalents to the process pair "checkpoint messages" are simply the retryable intentions written not-in-place to the "desktop" and then idempotence allows consistent duplication of uncompleted effort on takeover. All virtual-addressed variables on the process stack would be lost on restart, unless that becomes globally addressable and non-volatile in the future (and not zeroed out after machine check, panic, etc.), in which case "Update in place" (Jim Gray) will still remain "the poison apple," and a problem for stack recovery after failure.

Database Queries in the Browser Front End

Slabs that contained their base + offset data and the C++ source code for accessing that data could be fetched into a browser, compiled and executed in a sandbox (using the Windows 10 Sandbox, or the Linux Secure Computing Mode - (seccomp).) After the fraction of a second that takes, whatever queries were desired could be accessed in microseconds, even when the browser's device was disconnected due to non-payment of the cellphone bill, or too far away from a cell tower, or the internet was disrupted, or the database service had an outage, or one of the cloud systems containing a facility needed for remote access was disrupted, or there is a major cyberattack, or a flooding disaster, or … basically in spite of anything that happens anywhere else, but keep a charger handy.

Using the current code base with little modification, that should be possible now, so that's not really a future. It does bear repeating, however, since that list of misfortunes is not getting less likely.

Remote JoinedIndex Class Objects

It is also a small matter of software to provide some version of a remote mmap service that could allow remote access to these objects (outside of the datacenter Gen-Z fabric or high-speed DFS access). Naively done, that would have obvious performance problems the first time one performed a binary search of a really large index over distance: each binary hop would require a remotely accessed cache line fetch over Gen-Z, or a memory page fetch over DFS. That would only occur once for a memoized joined index fetching a range of results, but then for processing the result, every fetched resulting element joined row would by inference access each table data for that row, and for each table row in the joined row result there would be another random access with another

speed of light delay. That would not please the consumers using such a service. Could this problem be solved?

The initial remote range query would have to be overcome, by a derived class of **JoinedIndex** front-ending access to the base class mmapped data of a memoized joined row array. The **RemoteJoinedIndex** objects would look exactly like the base objects but only contain 1 in 2^{10} or so of the elements: a cut-down class that could be transmitted easily over the network in a slab containing the code. The cut-down factor could be statically deduced from the size of the base object: the idea is to have a sequence gap locally corresponding remotely to a contiguous range of memoized joined row index to process in the memory-centric system at the datacenter. Then you find the gap locally with a range query and process the range remotely. Potentially those gaps could lead to partitioned (not the disconnections kind of partition) services to distribute the query load at scale in the manner of distributed hashing, but in this case even balance would be provided by the gap mathematics and you would never have skew. Taking a leaf from the pages of the playbook of DataSeries, the returned rowsets would be compressed and contain enough state to preclude further random access fetches to this specific slab.

There is some discussion (Hashing versus NLJ) on the topic of nested loop join (the NLJ) versus hash joins. NLJ steps into a contiguous range of values, in this case contiguous in the binary sense, which seed the next level of joining easily. Hashing wins when the database is enormous and point queries dominate over range queries. Linear hashing is a new development merging the best of both. Not being a database expert and merely a transaction system developer, this seems like a lot of bother. Memoized joins cut through the Gordian knot of this problem, even the distributed remote access problem.

Remembering the comment at the end of the Synopsis, "the performance optimization provided by the database 'project' operation is unnecessary, except on display operations," this was a truthful statement inside the memory-centric system at the datacenter due to inferential access by pointer reference to all the soft-defined metadata concerning the `JoinedRowStruct`, which is a barren array of `int` K-values. However, it is not true for remote access, so we need to project out of that memoized joined row array, which of the possible columns of tables participating in the join that each returning **PackedRowString** (along with **JoinedIndex** and **RemoteJoinedIndex**, another as-of-yet unwritten class) element's columns are populated with. Fortunately, there is an `enum class Column` with a constant `enum` for every column in every table in the relational database in the current code base to identify these and a member function `inline char* columnStr(Column columnEnum)` on the **RowString** class which, when indexed by the K-value in the memoized joined row array, will return a pointer to that null terminated column string value. The request on the user side would identify (or know by inference) the memoized joined row array instance, and

have request value pairs of [`columnEnum`, and `packDepth` (sufficient to identify the instance of the table for the row-column to be projected in the memoized joined row struct since the same table may be visited many times in the join. `packDepth==0` would have two tables, from-table and to-table, but infinite recursion would happen if they were the same ambiguous `columnEnum`).]

Packing those into a **PackedRowString** element would then be done by assembling an array of `char*` pointers the same size as the array of projected columns in the remote request and doing the compile time recursion across the **RelationVector** parameter pack to fill that array with `char*` pointers into the database, in the projection requested order, with the column values of the table as they are encountered recursively at the appropriate `packDepth`. After the recursion, any pointers in the array with a null signifies an error (since SQL nulls are not supported), and you walk the array, in order, to fill the **PackedRowString** element with column null-terminated strings. This should take nanoseconds per **PackedRowString** element in the returning rowset. It is possible that a single service could saturate the NIC returning requested packed rowsets.

In theory, since these mmapped slabs, WAL logs and now Bloom filters can be replicated, the databases of the world in the form of mmapped slab and log-trail + BF replicas could be spread out to internet points of presence (PoP) and collocated there in Gen-Z memory-centric hubs. Datacenters would then shrink dramatically in size.

Internet of things and large non-volatile memories

First Hinix and then Western Digital stubbed their toes on memristor crossbar fabrication and broke Stanley Williams, HP and HPE's collective hearts. However, the memristor mantra "Ions for storage!" is still the future. This remains true since Heisenberg Uncertainty approximates to $\Delta\chi\Delta\rho \geq \frac{\hbar}{2}$ and the mass of an ion such as Gallium (normally found with a +3 oxidation state, occasionally +1) is 56,501 times the mass of an electron. Therefore, storing memory using electrons (their charge, their electric field, or magnetic field, or their resistance) into a confined small surface will have at least 100 times less potential spatial density than confining groups of ions, or the holes they leave behind when they transit, for storage in some solid state matrix.

This is simplistic logic and imprecise, but the power of uncertainty cannot be denied: electrons apparently don't like to be localized for a measurement, beyond current DRAM levels, and ions don't seem to mind being crowded while giving up their information. The problem is getting the bit information out of a 100^3 denser matrix than electrons like to sense, using electronic circuits. Hence, we are all waiting for some whiz kid to discover a method of bypassing the electronics via decoherence of an entangled block of ions into an optical numeric signal reversibly! She's out there, somewhere.

It is inevitable that at some technological point we will see the advent of huge and fast nonvolatile memory (NVM) in personal devices.

Whenever huge density NVM in portable sizes becomes available, it could be deployed already containing large amounts of data, not just the Library of Congress, but large databases not requiring download for queries of enormous complexity at the speed of binary code execution. The equivalent of many Wikipedias of geographic database deduction instantly available at extremely low power and cooling cost.

Components of larger assemblies could know their place not only in the assemblies, but the environments in which they dwell. Completely disconnected devices could possess a priori knowledge of all the kinds of devices that they might interact with. No more dumb devices. “Plug and play” and monitor and interact on a galactic scale. Every single component would know its history of connectedness with everything else. Every component would know its utilization history and could profile that versus any other component or assembly of components that it might come in contact with. Nothing would need to be tethered to data centers or to share private data with data centers to calculate anything, because every component would be knowledgeable and could do its own querying locally. Everything would know everything and be able to do the most complex queries on that knowledge at the cost of a few instructions: with lower energy use and heat, all the answers produced in a handful of microseconds.

Running the Current Code Base on Big Hardware with 100 TB Database

What is needed next is a large memory system of 100s of Terabytes and a much larger and more complex set of databases to test the scalability of this system out and do the parallelization studies on. That system should have 50 or 100 table join queries so as to create really large memoized joined row arrays and indexes, which would be a more strenuous test of the C++ compiler than of anything else, because the components of the current code base scale remarkably well, and they will stand up to the challenge. [Faster sort algorithms are available, the simple ones were deployed here, because they were easy to instrument.] Since the construction of the database tables and indexes, and the memoized joined row arrays and indexes, can be staged in parallel using single writer mmap processes, this will likely work well on a DFS mmap, but it would be of greater interest to see this working on a giant Gen-Z system or a decent simulation of one, like a big HPE Superdome Flex with 48 TB of memory.

13. Summary

The `JoinedRowStruct` that is the heart of the relational database system in the current demo code base is remarkably simple in its definition, but has massive and detailed semantic expression hidden in the soft-definition drawn by the **JoinedRow** template class:

```
struct JoinedRowStruct
{
    int k[arrayCount];
};
```

That is the power of inference at work. Inference is the moral equivalent to quantum entanglement's "spooky action at a distance," but with Einstein's "hidden variables" developed within the type system of C++, which "doesn't play dice."

Quantum entanglement has stood up to experiments from Einstein's first challenges, but quantum mechanics still remains to be incomplete, due to the existence of 100s of interpretations related to problems with the observer not described by QM and issues around decoherence not described by QM. Some of those interpretations (Bohm, others) are similar to Einstein's view of preexisting "hidden variables" using pilot waves and countering the "it doesn't exist until the observer makes the observation" view from Copenhagen.

All of the rows and columns, the tables and indexes, the relations and relation vectors defining the schema of the nested loop join that produces a consistent set of joined rows in the output of a query: all of those are inferences are produced collectively and consistently by the inference engine known as the C++11 compiler, specifically here, g++ (gcc 4.8.5) and that is done almost completely at compile time, not burdening runtime performance in the inner loop. Preexisting "hidden variables."

The actual meaning of the query and its results that is stored in the tables (truth tables) in rows (truthful propositions) and their intended relationships, that meaning arises either correctly or falsely according to the actual data contained in the .csv files and the intent of the relationships between the columns represented in those files. The actual programs doing the queries have no meaning per se, that meaning is a derivation wholly arising from the data. However, if the data is true and the data relationships precise and accurate to something in the real world, then the deductions produced by the programs in the current demo code base will be truthful, precise and accurate, and those deductions will execute at the maximum speed possible, because of the generic programming inference engine in C++11. Similarly, in the world at large supported by quantum entanglement, the meaning of it all is us, the numbers living in it.

Update on Compiler Versions Supported by the Current Code Base

Just before going to press, the current code bas running on Centos-7 g++ (gcc 4.8.5) had a small amount of refactoring to get it to compile cleanly and run on Fedora 28 g++ (gcc 8.2.1) on an older small laptop not worthy of reporting performance numbers. The code compiles cleanly (no warnings) for C++11, C++14, C++17 and C++2a, which is the experimental version of the coming C++20. The 2 microsecond double route joins went up to 3 microseconds on the laptop.

A Caveat for C++

This kind of performance improvement (3-6 orders of magnitude improvement in producing complete result sets from large complex queries), which is unattainable by means outside of C++11 and its later versions, is clearly disruptive of current software laboring under lesser means (SQL or NoSQL, dynamic languages wrapping SQL or NoSQL language queries, systems based upon them, etc.)

One person's victorious disruption is another person's extinction event.

Viewing that disruption as an inevitable and irreversible extinction event is a natural conclusion for the large database corporations whose intellectual property is on the line, since the C++11, C++14 and C++17 compiler is already available in wide release on all Linux, Windows and AOS systems.

There would be a risk of forestalling that inevitability, by embracing and coopting future C++11, 14, 17, 20… development in these nascent and fragile areas of the variadic template parameter pack recursion, static globals, const and constexpr rules, by the powerful interests with a lot at stake in maintaining the status quo, were gcc and g++ not in the hands of the Free Software Foundation, GNU and Richard Stallman: bless their collective and battle-hardened hearts. They can, if they are vigilant, keep that misfortune from preventing the efficiency improvements and the resulting reductions in energy demand that the world has a right and a need to see deployed in the data centers of the world. They can, if they maintain both desire and vigilance, keep a path open for this kind of programming to exist in the future, if not some refactored or forked version of the current demo code base.

Gratitude and Appreciation

There is a large and extended community of people, whose inspiring work, or mentoring and support led to this work over several decades. They were critical to this accomplishment and deserve gratitude and appreciation:

- USF Physics Research (NASA, DOE): Dr. Richard Henke, Ronald Cassou, and Dr. Allan Frank, led by Dr. Eugene V. Benton, and aided by Dr. Wolfgang Heinrich and Dr. Dietmar Hildebrand. Completely automated stepper-stage microscope scanning systems running on hybrid analog-digital Imanco computers connected by a LAN pair of DR-11Cs and RT/RSX drivers in 1981. Astounding.
- The Israeli Air Force MIRS/MADA project (Tel Aviv): Joanne Wheeler, led by Lt. Colonel Elie Ben-Dan and George Mahelona (CACI-London), aided by Jake Greenwell and Lance Carnes (CACI-Alameda). They got the NonStop relational database systems working using broken TMF1. Amazing.
- NASA KSC-Systems Integration Branch-Space Station and STS Integration and Test: Victor Lewis (PRC-GIS,) Bronwen Chandler, Larry Morgan (NASA Lead Engineer,) and led by Jan Heuser (NASA Chief.) Launching shuttles and orbiting space stations using ancient mainframes to do the LPS GOAL language builds running in TTL-based 16-bit consoles connected together by a 64KB TTL crossbar memory switch called the Common Data Bus holding all the distributed sensor and effector data in condensed form from across Cape Canaveral. It's unbelievable that the STS got the ISS put together, based upon that infrastructure, and the ISS is still flying, in spite of the loss of one shuttle ascending and one descending from orbit.
- Tandem NonStop Computers, led by Jimmy Treybig and ruled by Wendy Bartlett, now HPE NonStop led by Andrew Bergholz - the most amazing system that should have taken over the world:
    - Language and Tools: Roland Findlay, Keith Evans (Pathway), Mark Molloy, David Cutler, Kristi Andrews, Ron Tischler, Billy Jasiniecki, Geoff Baldwin, Paul Del Vigna (Lead Developer), led in the past by Ann Sophie-Dean.
    - NonStop TMF Group (distributed and clustered transactions): Pat Helland, Shel Finkelstein, Jimbo Lyon, Matt McCline, Jim Carley, Steve Pearson, Ron Cassou, Shang-Shen Tung, Albert Gondi, Johannes Klein, Sitaram V. Lanka, Malcolm Mosher, Jr., Larry Emlich, Robert Lennie, Muhammad Shafiq, Yu-Cheung Cheung, Trina R. Wisler, Jim B. Tate, Jim Willis, David Hege, David Wisler, Greg Stewart and Raghunath Chandra, led early on by Mala Chandra, Dr. David Holt and later by Rikki Westerschulte.
    - NonStop SQL Group: Professor Jim Gray (the Rector of Tandem University), Franco Putzolu, Donald Slutz, Gary S. Smith, Donald Maier, Robbert C. Van Der Linden, Pedro Celis, Michael J. Skarpelos, Stan Kozinski, Ram Kosuru, led in the past by Roberta Henderson and later Bob Taylor.

    - NonStop Kernel O/S: James Katzman, Joel Bartlett, Richard Carr, Srinivasa D. Murthy, Srinivas Brahmaroutu, Pankaj Mehra, Lars Plum, Mike Rice, led now by Anne Bird.
- HP/HPE Labs during "The Machine" era: Alistair Veitch, Craig Soules, Charles "Brad" Morrey, Kim Keeton, Harumi Kuno, Goetz Graefe, Joe Tucek, Eric Anderson, Mark Lillibridge, Lucy Cherkasova, Jay Wylie, Jonas Arndt, Jim Baker, Hideaki Kimura, Haris Volos, Yupu Zhang, John Byrne, Terence Kelly, Dhruva Chakrabarti, Hans Boehm, Alvin AuYoung, Stan Park, Kumud Bhandari, Milind Chabbi, Zhuan Chen, Tuan Tran, Mesut Kuscu, Onkar Patil, Dan Feldman, Rob Schreiber, Antonio Lain, Ram Swaminathan, Kave Eshghi, Mehran Kafai, Drew Walton, Paolo Faraboschi, Derek Sherlock, Qiong Cai, Al Davis, Amit Sharma, Ron Noblett, Sarah Silverthorn, Sharad Singhal, Preetam Pagar, Keith Packard, Steve Geary, Rocky Craig, Betty Dall, Drew Walton, Steve Lyle, Tim Forelle, Kevin Wilkinson, Alkis Simitsis, Dwight Riley, led by Partha Ranganathan, Richard Friedrich, Jaap Suermondt, John Sontag, Martin Fink, Cullen Bash and Mark Potter. Memory-centric computing will take over the world in the form of Gen-Z.
- The Free Software Foundation: Richard Stallman, Eben Moglen, Bradley M. Kuhn, et al. Without the FSF keeping the wolves at bay, the software industry would be at their mercy.
- The C++ language and compiler standards ISO C++ committee called WG21, officially ISO/IEC JTC1 (Joint Technical Committee 1) / SC22 (Subcommittee 22) / WG21 (Working Group 21): They have enabled the development of an astonishing purely inferential software system in C++11 and later.
- The army of developers creating and supporting gcc and g++ at GNU: they have done a brilliant job on puny resources.
- The army of developers and writers at http://stackoverflow.com: without their clearing away the confusion, this work would never have progressed beyond the early stage.
- The army of programmers who wrote the C++11 version of tuple: it is unreadable code and creates problems while solving problems, but it got the job done in an utterly indispensable way. Nothing else that attempts to do what <tuple> does, works properly.
- The Holenstein brothers of Gravic™ and Bill Highleyman: they have taken log-reading to a phenomenal level and raised the bar of fault tolerant distributed reliable computing.
- Deep thanks to my collaborator Yanpei and his family, my dear friends Phu and Koko, and to Susie, the muse who inspired me and helped me to accomplish this effort.
- And finally, my deepest thanks and appreciation go to the 13th-century Japanese Lotus Sutra Buddhist priest Nichiren and my mentor Daisaku Ikeda: I follow the former at the direction of the latter, who opened the door, showed it to me, caused me to awaken to it, and induced me to enter into it [一大事].

Appendix – Demo programs output

There are four demo programs in the Kaldanes GitHub code base:
https://github.com/charlesone/Kaldanes

They are all C++11 (g++ 4.8.5) Centos-7 Linux console programs spinning output line-by-line to the console, although they now compile cleanly and run on C++11, C++14, C++17 and C++2a (g++ 8.2.1.) They all do performance analysis using the precision nanosecond clock support from C++11. They have not yet been run on Windows or AOS.

SortDemo output

SortDemo compares the four string classes (std::string, **Direct**, **Symbiont** and **Head**) using identical random strings in identical arrays for apples to apples comparison: that is why the number of compares and swaps for both quick sort and merge sort are listed, the compares and swaps for the same run should be precisely identical using the same sort code for the same strings, whatever the class that is used. Note that for std::strings in the million strings merge sort the time is independent of string length. Under the working hypothesis on why std::string is quadratic, the object initialization in the temporary arrays is the problem and those temporary arrays hold some metadata requiring default initialization for every array element, whether it is used or not.

```
[charles@localhost Kaldanes]$ ./SortDemo


Three different string types, identical in content and
sorting logic, but not in timings.

[Note that the number of swaps and compares for identical arrays,
   using the identical code: these statistics should be identical.

 Remember to set the C++11 switch in the IDE or compiler!

 Remember to use release builds if you are analyzing performance,
   otherwise Symbionts will be very slow!

 Remember to set the "ulimit -s" soft and hard stack limits to unlimited,
   otherwise it can die!]


Quick Sort 100 10-byte strings
Direct: 30.024 microseconds, 271.000 swaps, 838.000 compares
Symbiont: 24.894 microseconds, 271.000 swaps, 838.000 compares
Head: 23.897 microseconds, 271.000 swaps, 838.000 compares
std::string: 44.323 microseconds, 271.000 swaps, 838.000 compares

Merge Sort 100 10-byte strings
Direct: 35.836 microseconds, 625.000 swaps, 540.000 compares
Symbiont: 26.725 microseconds, 625.000 swaps, 540.000 compares
Head: 26.357 microseconds, 625.000 swaps, 540.000 compares
std::string: 73.710 microseconds, 625.000 swaps, 540.000 compares

Quick Sort 100 100-byte strings
Direct: 43.777 microseconds, 267.000 swaps, 835.000 compares
```

Symbiont: 27.890 microseconds, 267.000 swaps, 835.000 compares
Head: 51.228 microseconds, 267.000 swaps, 835.000 compares
std::string: 38.394 microseconds, 267.000 swaps, 835.000 compares

Merge Sort 100 100-byte strings
Direct: 48.071 microseconds, 614.000 swaps, 544.000 compares
Symbiont: 27.232 microseconds, 614.000 swaps, 544.000 compares
Head: 28.717 microseconds, 614.000 swaps, 544.000 compares
std::string: 72.111 microseconds, 614.000 swaps, 544.000 compares

Quick Sort 100 1000-byte strings
Direct: 132.921 microseconds, 266.000 swaps, 843.000 compares
Symbiont: 25.087 microseconds, 266.000 swaps, 843.000 compares
Head: 33.784 microseconds, 266.000 swaps, 843.000 compares
std::string: 50.519 microseconds, 266.000 swaps, 843.000 compares

Merge Sort 100 1000-byte strings
Direct: 206.671 microseconds, 624.000 swaps, 544.000 compares
Symbiont: 24.102 microseconds, 624.000 swaps, 544.000 compares
Head: 26.835 microseconds, 624.000 swaps, 544.000 compares
std::string: 67.385 microseconds, 624.000 swaps, 544.000 compares

Quick Sort 10000 10-byte strings
Direct: 4417.937 microseconds, 42448.000 swaps, 181933.000 compares
Symbiont: 4046.971 microseconds, 42448.000 swaps, 181933.000 compares
Head: 3296.360 microseconds, 42448.000 swaps, 181933.000 compares
std::string: 6191.679 microseconds, 42448.000 swaps, 181933.000 compares

Merge Sort 10000 10-byte strings
Direct: 4379.722 microseconds, 128145.000 swaps, 120469.000 compares
Symbiont: 3521.941 microseconds, 128145.000 swaps, 120469.000 compares
Head: 3065.058 microseconds, 128145.000 swaps, 120469.000 compares
std::string: 111599.644 microseconds, 128145.000 swaps, 120469.000 compares

Quick Sort 10000 100-byte strings
Direct: 3275.111 microseconds, 42586.000 swaps, 182638.000 compares
Symbiont: 2567.446 microseconds, 42586.000 swaps, 182638.000 compares
Head: 2404.953 microseconds, 42586.000 swaps, 182638.000 compares
std::string: 3527.660 microseconds, 42586.000 swaps, 182638.000 compares

Merge Sort 10000 100-byte strings
Direct: 3801.491 microseconds, 128142.000 swaps, 120413.000 compares
Symbiont: 2374.620 microseconds, 128142.000 swaps, 120413.000 compares
Head: 2020.152 microseconds, 128142.000 swaps, 120413.000 compares
std::string: 76728.072 microseconds, 128142.000 swaps, 120413.000 compares

Quick Sort 10000 1000-byte strings
Direct: 10339.625 microseconds, 42121.000 swaps, 187359.000 compares
Symbiont: 2263.947 microseconds, 42121.000 swaps, 187359.000 compares
Head: 2166.872 microseconds, 42121.000 swaps, 187359.000 compares
std::string: 5259.493 microseconds, 42121.000 swaps, 187359.000 compares

Merge Sort 10000 1000-byte strings
Direct: 34495.742 microseconds, 128094.000 swaps, 120561.000 compares
Symbiont: 2794.445 microseconds, 128094.000 swaps, 120561.000 compares
Head: 1983.078 microseconds, 128094.000 swaps, 120561.000 compares
std::string: 78939.452 microseconds, 128094.000 swaps, 120561.000 compares

Quick Sort 1000000 10-byte strings
Direct: 281715.578 microseconds, 5804008.000 swaps, 26523855.000 compares
Symbiont: 275304.609 microseconds, 5804008.000 swaps, 26523855.000 compares
Head: 225242.776 microseconds, 5804008.000 swaps, 26523855.000 compares
std::string: 669127.282 microseconds, 5804008.000 swaps, 26523855.000 compares

warning: the std::string merge sorts of 1 million records can take 6740 times longer!
(30 minutes or so on my i5-3570 CPU)

Merge Sort 1000000 10-byte strings
Direct: 333769.189 microseconds, 19346100.000 swaps, 18674064.000 compares
Symbiont: 286190.343 microseconds, 19346100.000 swaps, 18674064.000 compares

```
Head: 258762.763 microseconds, 19346100.000 swaps, 18674064.000 compares
std::string: 1703180037.844 microseconds, 19346100.000 swaps, 18674064.000 compares

Quick Sort 1000000 100-byte strings
Direct: 500382.234 microseconds, 5804134.000 swaps, 26612525.000 compares
Symbiont: 455411.520 microseconds, 5804134.000 swaps, 26612525.000 compares
Head: 298926.255 microseconds, 5804134.000 swaps, 26612525.000 compares
std::string: 718701.456 microseconds, 5804134.000 swaps, 26612525.000 compares

Merge Sort 1000000 100-byte strings
Direct: 748486.220 microseconds, 19346451.000 swaps, 18674253.000 compares
Symbiont: 615096.744 microseconds, 19346451.000 swaps, 18674253.000 compares
Head: 281864.353 microseconds, 19346451.000 swaps, 18674253.000 compares
std::string: 1703700587.318 microseconds, 19346451.000 swaps, 18674253.000 compares

Quick Sort 1000000 1000-byte strings
Direct: 1907102.428 microseconds, 5756390.000 swaps, 27144246.000 compares
Symbiont: 516912.891 microseconds, 5756390.000 swaps, 27144246.000 compares
Head: 296771.184 microseconds, 5756390.000 swaps, 27144246.000 compares
std::string: 1032007.470 microseconds, 5756390.000 swaps, 27144246.000 compares

Merge Sort 1000000 1000-byte strings
Direct: 6241556.331 microseconds, 19346660.000 swaps, 18674968.000 compares
Symbiont: 719999.598 microseconds, 19346660.000 swaps, 18674968.000 compares
Head: 282007.334 microseconds, 19346660.000 swaps, 18674968.000 compares
std::string: 1748086946.445 microseconds, 19346660.000 swaps, 18674968.000 compares
[charles@localhost Kaldanes]$
```

TableDemo output

The **TableDemo** program builds a database from scratch, by loading three .csv files (created from .dat files obtained from OpenFlights.org) into **RowString** arrays (tables) as automatic variables on the call stack, then building ten **IndexString** arrays (indexes) on columns of those tables and then using increasingly complex queries on airlines, airports and routes, which results in the creation of a **QueryPlan** object from a triplet of **RelationVector** types and corresponding format parameter objects and processing nested loop joins into a **JoinedRow** output array and printing the results of all those queries as **JoinedRow** objects as quadruplets (source airport, route, destination airport, airline). All of this in less than 1/8 of a second. Finally, it builds an improper **QueryPlan** object which demonstrates the query plan checker by throwing a linkage error exception, which is uncaught and ends the program fatally.

```
[charles@localhost Kaldanes]$ ./TableDemo


[Note: Remember to set the C++11 switch in the IDE or compiler!

 Remember to use release builds if you are analyzing performance,
   otherwise Index builds will be very slow!

 Remember to set the "ulimit -s" soft and hard stack limits to unlimited,
   otherwise it can die!]


Time for space allocating and loading the three tables from CSV files = 29748.446 microseconds
Time for space allocating the ten indexes = 0.676 microseconds
Time for copying the keys and sorting the ten indexes = 91222.293 microseconds
Time for each index on the average = 9122.229 microseconds

First, let's execute a point query lambda on the airportsName index, for an airport that exists, and one that is imaginary.
```

Time for a point query on an index on a smaller table = 5.976 microseconds

"La Guardia Airport" Airport record query:
Returns: 3360 (Existing)

11743,La Grande-4 Airport,La Grande-4,Canada,YAH,CYAH,53.754699707,-73.6753005981,1005,\N,\N,\N,airport,OurAirports
11102,La Grande/Union County Airport,La Grande,United States,LGD,KLGD,45.2901992798,-118.007003784,2717,-7,A,\N,airport,OurAirports
3697,La Guardia Airport,New York,United States,LGA,KLGA,40.77719879,-73.87259674,21,-5,A,America/New_York,airport,OurAirports
5811,La Isabela International Airport,La Isabela,Dominican Republic,JBQ,MDJB,18.572500228881836,-69.98560333251953,98,-4,U,America/Santo_Domingo,airport,OurAirports
6050,La Jagua Airport,GarzÃ³n,Colombia,GLJ,SKGZ,2.1464,-75.6944,2620,-5,U,America/Bogota,airport,OurAirports
---

Time for a point query on an index on a smaller table = 0.959 microseconds

"La Nunca Airport" Airport record query:
Returns: -3368 (Missing)

5786,La MÃ´le Airport,La MÃ´le,France,LTT,LFTZ,43.20539855957031,6.48199987411499,59,1,E,Europe/Paris,airport,OurAirports
2733,La Nubia Airport,Manizales,Colombia,MZL,SKMZ,5.0296,-75.4647,6871,-5,U,America/Bogota,airport,OurAirports
2847,La Orchila Airport,La Orchila,Venezuela,\N,SVLO,11.80720043182373,-66.17960357666016,5,-4,U,America/Caracas,airport,OurAirports
1053,La Palma Airport,Santa Cruz De La Palma,Spain,SPC,GCLA,28.62649917602539,-17.755599975585938,107,0,E,Atlantic/Canary,airport,OurAirports
6052,La Pedrera Airport,La Pedrera,Colombia,LPD,SKLP,-1.32861,-69.5797,590,-5,U,America/Bogota,airport,OurAirports
---

Second, let's execute a range query lambda on the routesAirlineId index, printing all Allegiant Airlines (AAY, ID=35) routes.

Time for a range query on an index on the biggest table (70K records) = 2.493 microseconds

"35" Routes record range query:
Returns: 36680 (Existing)

G4,35,AZA,6505,BZN,4020,,0,M80
G4,35,AZA,6505,CID,4043,,0,M80 319
G4,35,AZA,6505,DLH,3598,,0,M80
G4,35,AZA,6505,BLI,3777,,0,M80
G4,35,AVL,4007,PIE,3617,,0,320
G4,35,AVL,4007,PGD,7056,,0,M80
G4,35,AZA,6505,BIS,4083,,0,M80
G4,35,AZA,6505,EUG,4099,,0,M80
G4,35,AZA,6505,GRI,5740,,0,M80
G4,35,AZA,6505,GRR,3685,,0,319
G4,35,AZA,6505,GTF,3880,,0,M80

… [more of the same, edited for sanity.]

G4,35,LAS,3877,FCA,4127,,0,M80
G4,35,LAS,3877,FSD,4009,,0,M80
G4,35,LAS,3877,FAT,3687,,0,M80
G4,35,LAS,3877,EUG,4099,,0,M80
G4,35,LAS,3877,COS,3819,,0,M80
G4,35,LAS,3877,CID,4043,,0,M80
G4,35,LAS,3877,BZN,4020,,0,M80
G4,35,LAS,3877,CPR,3872,,0,M80
G4,35,LAS,3877,DSM,3729,,0,M80
G4,35,LAS,3877,DLH,3598,,0,M80

Third, let's make three relation vectors with 'from' and 'to' indexes for each, and see that we can print some table rows from there.

Relation Vector airportId2RouteSourceAirportId.fromIndex[24]:

24,St. Anthony Airport,St. Anthony,Canada,YAY,CYAY,51.3918991089,-56.083099365200006,108,-3.5,A,America/St_Johns,airport,OurAirports
---
Relation Vector airportId2RouteSourceAirportId byte size: 16

Relation Vector routeAirlineId2AirlineId.toIndex[15]:

15,Abelag Aviation,\N,W9,AAB,ABG,Belgium,N

Time for creating the three relation vectors = 0.909 microseconds

Fourth, let's make a tuple of three relation vectors that will form a join, and see that we can print the same table rows from there.

Time for creating the tuple containing the three relation vectors = 0.448 microseconds

Routes Join relation tuple<0> 'from' airportId2RouteSourceAirportId.fromIndex[24] (same as above):

24,St. Anthony Airport,St. Anthony,Canada,YAY,CYAY,51.3918991089,-56.083099365200006,108,-3.5,A,America/St_Johns,airport,OurAirports
---

Routes Join relation tuple<2> 'to' routeAirlineId2AirlineId.toIndex[15] (same as above):

15,Abelag Aviation,\N,W9,AAB,ABG,Belgium,N

Routes Join relation tuple byte size: 48

Fifth, let's make a linked (and checked) QueryPlan of the joined row tables from the relation vectors (which makes a tuple called relVecsTuple), and see if we can access a madeup joined route (SJC->LAS) from the JoinedRow.

Time for creating the query plan to join the three relation vectors = 2.561 microseconds

Query Plan for routes byte size: 8

Time for creating the joined row array to contain the join output = 0.520 microseconds

---
3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
AA,24,FAT,3687,PHX,3462,Y,0,CR9 CRJ
3462,Phoenix Sky Harbor International Airport,Phoenix,United States,PHX,KPHX,33.43429946899414,-112.01200103759766,1135,-7,N,America/Phoenix,airport,OurAirports
24,American Airlines,\N,AA,AAL,AMERICAN,United States,Y

Routes JoinedRow byte size: 16

Sixth, let's try a nested loop join listing all the valid air routes (having valid airlines, in or out airports) out of the Fresno Yosemite Airport

Time for doing the four table join = 42.293 microseconds

return count = 20

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
AA,24,FAT,3687,PHX,3462,Y,0,CR9 CRJ
3462,Phoenix Sky Harbor International Airport,Phoenix,United States,PHX,KPHX,33.43429946899414,-112.01200103759766,1135,-7,N,America/Phoenix,airport,OurAirports
24,American Airlines,\N,AA,AAL,AMERICAN,United States,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
AA,24,FAT,3687,LAX,3484,Y,0,CRJ
3484,Los Angeles International Airport,Los Angeles,United States,LAX,KLAX,33.94250107,-118.4079971,125,-8,A,America/Los_Angeles,airport,OurAirports
24,American Airlines,\N,AA,AAL,AMERICAN,United States,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
AA,24,FAT,3687,DFW,3670,,0,M80
3670,Dallas Fort Worth International Airport,Dallas-Fort Worth,United States,DFW,KDFW,32.89680099487305,-97.03800201416016,607,-6,A,America/Chicago,airport,OurAirports
24,American Airlines,\N,AA,AAL,AMERICAN,United States,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
AS,439,FAT,3687,SEA,3577,,0,CR7
3577,Seattle Tacoma International Airport,Seattle,United States,SEA,KSEA,47.44900131225586,-122.30899810791016,433,-8,A,America/Los_Angeles,airport,OurAirports
439,Alaska Airlines,\N,AS,ASA, Inc.,ALASKA,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
AS,439,FAT,3687,SAN,3731,Y,0,DH4
3731,San Diego International Airport,San Diego,United States,SAN,KSAN,32.7336006165,-117.190002441,17,-8,A,America/Los_Angeles,airport,OurAirports

439,Alaska Airlines,\N,AS,ASA, Inc.,ALASKA,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
UA,5209,FAT,3687,LAS,3877,Y,0,EM2
3877,McCarran International Airport,Las Vegas,United States,LAS,KLAS,36.08010101,-115.1520004,2181,-8,A,America/Los_Angeles,airport,OurAirports
5209,United Airlines,\N,UA,UAL,UNITED,United States,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
UA,5209,FAT,3687,DEN,3751,Y,0,CR7 CRJ
3751,Denver International Airport,Denver,United States,DEN,KDEN,39.861698150635,-104.672996521,5431,-7,A,America/Denver,airport,OurAirports
5209,United Airlines,\N,UA,UAL,UNITED,United States,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
AS,439,FAT,3687,PDX,3720,Y,0,DH4
3720,Portland International Airport,Portland,United States,PDX,KPDX,45.58869934,-122.5979996,31,-8,A,America/Los_Angeles,airport,OurAirports
439,Alaska Airlines,\N,AS,ASA, Inc.,ALASKA,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
F9,2468,FAT,3687,DEN,3751,,0,319
3751,Denver International Airport,Denver,United States,DEN,KDEN,39.861698150635,-104.672996521,5431,-7,A,America/Denver,airport,OurAirports
2468,Frontier Airlines,\N,F9,FFT,FRONTIER FLIGHT,United States,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
Y4,5325,FAT,3687,GDL,1804,,0,320
1804,Don Miguel Hidalgo Y Costilla International Airport,Guadalajara,Mexico,GDL,MMGL,20.521799087524414,-103.31099700927734,5016,-6,S,America/Mexico_City,airport,OurAirports
5325,Volaris,\N,Y4,VOI,VOLARIS,Mexico,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
AS,439,FAT,3687,LAX,3484,Y,0,CRJ
3484,Los Angeles International Airport,Los Angeles,United States,LAX,KLAX,33.94250107,-118.4079971,125,-8,A,America/Los_Angeles,airport,OurAirports
439,Alaska Airlines,\N,AS,ASA, Inc.,ALASKA,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
G4,35,FAT,3687,LAS,3877,,0,M80
3877,McCarran International Airport,Las Vegas,United States,LAS,KLAS,36.08010101,-115.1520004,2181,-8,A,America/Los_Angeles,airport,OurAirports
35,Allegiant Air,\N,G4,AAY,ALLEGIANT,United States,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
UA,5209,FAT,3687,LAX,3484,Y,0,CR7 EM2
3484,Los Angeles International Airport,Los Angeles,United States,LAX,KLAX,33.94250107,-118.4079971,125,-8,A,America/Los_Angeles,airport,OurAirports
5209,United Airlines,\N,UA,UAL,UNITED,United States,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
DL,2009,FAT,3687,SLC,3536,Y,0,CR9 CRJ
3536,Salt Lake City International Airport,Salt Lake City,United States,SLC,KSLC,40.78839874267578,-111.97799682617188,4227,-7,A,America/Denver,airport,OurAirports
2009,Delta Air Lines,\N,DL,DAL,DELTA,United States,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
US,5265,FAT,3687,PHX,3462,Y,0,CR9 CRJ
3462,Phoenix Sky Harbor International Airport,Phoenix,United States,PHX,KPHX,33.43429946899414,-112.01200103759766,1135,-7,N,America/Phoenix,airport,OurAirports
5265,US Airways,\N,US,USA,U S AIR,United States,Y

3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-8,A,America/Los_Angeles,airport,OurAirports
DL,2009,FAT,3687,GDL,1804,Y,0,738
1804,Don Miguel Hidalgo Y Costilla International Airport,Guadalajara,Mexico,GDL,MMGL,20.521799087524414,-103.31099700927734,5016,-6,S,America/Mexico_City,airport,OurAirports
2009,Delta Air Lines,\N,DL,DAL,DELTA,United States,Y

```
3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-
8,A,America/Los_Angeles,airport,OurAirports
US,5265,FAT,3687,LAX,3484,Y,0,CRJ
3484,Los Angeles International Airport,Los Angeles,United States,LAX,KLAX,33.94250107,-118.4079971,125,-8,A,America/Los_Angeles,airport,OurAirports
5265,US Airways,\N,US,USA,U S AIR,United States,Y
-------------------------------------------------------------------------------------------------------------------------------
3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-
8,A,America/Los_Angeles,airport,OurAirports
UA,5209,FAT,3687,SFO,3469,Y,0,EM2 CRJ
3469,San Francisco International Airport,San Francisco,United States,SFO,KSFO,37.61899948120117,-122.375,13,-8,A,America/Los_Angeles,airport,OurAirports
5209,United Airlines,\N,UA,UAL,UNITED,United States,Y
-------------------------------------------------------------------------------------------------------------------------------
3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-
8,A,America/Los_Angeles,airport,OurAirports
AM,321,FAT,3687,GDL,1804,,0,738
1804,Don Miguel Hidalgo Y Costilla International Airport,Guadalajara,Mexico,GDL,MMGL,20.521799087524414,-103.31099700927734,5016,-
6,S,America/Mexico_City,airport,OurAirports
321,AeroMÃ„â€Ã‚Â©xico,\N,AM,AMX,AEROMEXICO,Mexico,Y
-------------------------------------------------------------------------------------------------------------------------------
3687,Fresno Yosemite International Airport,Fresno,United States,FAT,KFAT,36.77619934082031,-119.71800231933594,336,-
8,A,America/Los_Angeles,airport,OurAirports
US,5265,FAT,3687,DFW,3670,,0,M80
3670,Dallas Fort Worth International Airport,Dallas-Fort Worth,United States,DFW,KDFW,32.89680099487305,-97.03800201416016,607,-
6,A,America/Chicago,airport,OurAirports
5265,US Airways,\N,US,USA,U S AIR,United States,Y
-------------------------------------------------------------------------------------------------------------------------------


_____
Statistics for the run of the program, including:
  1. Allocating and loading three tables from CSV files on disk, around 80K rows.
  2. Allocating, copying the keys and sorting ten indexes on those three tables.
  3. Executing multiple point and range query lambdas on the indexes, as a test.
  4. Creating three relation vectors, each containing a from-index and a to-index.
  5. Creating a tuple from those three relation vectors to store the query objects.
  6. Creating and optimizing a database join query plan for those three relation vectors.
  7. Creating a joined row output array and doing a query plan nested loop join into it.

Total run time spent in main() procedure thus far = 0.12830 seconds

Total relational database system time spent in main() procedure thus far, including I/O = 0.12103 seconds

Total display output time spent in main() procedure thus far = 0.00727 seconds
_____

Finally, let's make a linked (and checked) QueryPlan of the joined row tables with one extra relation vector, that violates the linkage rule: that every from-index (after the first one) has a preceding to-indexed or initial from-indexed table to link from. The query plan checker should throw  an exception (exception name at end of error string).

Exception:
N9QueryPlanII14RelationVectorI11IndexStringIL6Column0EcLm186EL5Table0ELm14ELm1024ELm7EES1_ILS2_27EcLm65ELS3_2ELm9ELm1024ELm7EEES0_I
S1_ILS2_21EcLm404ELS3_1ELm8ELm1024ELm7EES1_ILS2_3EcLm186ELS3_0ELm14ELm1024ELm7EEES0_IS1_ILS2_29EcLm65ELS3_2ELm9ELm1024ELm
7EES4_ES0_IS1_ILS2_25EcLm65ELS3_2ELm9ELm1024ELm7EES1_ILS2_15EcLm404ELS3_1ELm8ELm1024ELm7EEEEE38Relation_Vector_Linkage_Rule_Vi
olationE
Segmentation fault (core dumped)
[charles@localhost Kaldanes]$
```

JoinDemo output

The **JoinDemo** program builds the same database as **TableDemo** with three slightly different **RelationVector** types and formal parameter objects in different join order from that created in **TableDemo** and twice performs an identical set of ten queries for the top passenger carriers in the airline industry, first as nested loop joins, and then as memoized joins. The memoized joins execute in microseconds since the output arrays created are small (ranging from 640 to 2340 joined row objects of 16 bytes each) and are around 2.5 orders of magnitude faster than the nested loop

joins (less than 10 microseconds versus multiple milliseconds). The output objects are quadruplets (airline, route, source airport, destination airport) and are counted, not printed, because there are too many.

```
[charles@localhost Kaldanes]$ ./JoinDemo


[Note: Remember to set the C++11 switch in the IDE or compiler!

 Remember to use release builds if you are analyzing performance,
   otherwise Index builds will be very slow!

 Remember to set the "ulimit -s" soft and hard stack limits to unlimited,
   otherwise it can die!]


Time for space allocating and loading the three tables from CSV files = 28689.818 microseconds.
Time for space allocating the ten indexes = 0.688 microseconds.
Time for copying the keys and sorting the ten indexes = 89496.312 microseconds.
Time for each index on the average = 8949.631 microseconds.
Time for creating the three relation vectors = 0.471 microseconds.
Time for creating the tuple containing the three relation vectors = 0.447 microseconds.
Time for creating the query plan to join the three relation vectors = 5.858 microseconds.
Time for creating the joined row array to contain the join output = 0.653 microseconds.
_____________________________________________________________________________

First, we do ten standard join queries listing the valid routes (possessing valid airlines, from and to airports) for the top ten passenger carriers in the world (as of 2016).

Time for American Airlines (199M passengers) join query = 2220.231 microseconds.
Using 4703 range queries.
Reporting 2340 valid air routes.

Time for Delta Air Lines (184M passengers) join query = 1808.416 microseconds.
Using 3962 range queries.
Reporting 1977 valid air routes.

Time for Southwest Airlines (152M passengers) join query = 915.955 microseconds.
Using 2291 range queries.
Reporting 1140 valid air routes.

Time for United Airlines (143M passengers) join query = 1864.007 microseconds.
Using 4358 range queries.
Reporting 2172 valid air routes.

Time for Ryanair/Ireland (120M passengers) join query = 2179.461 microseconds.
Using 4969 range queries.
Reporting 2482 valid air routes.

Time for China Southern Airlines (85M passengers) join query = 1211.766 microseconds.
Using 2894 range queries.
Reporting 1422 valid air routes.

Time for China Eastern Airlines (81M passengers) join query = 1032.598 microseconds.
Using 2502 range queries.
Reporting 1211 valid air routes.

Time for easyJet/UK (73M passengers) join query = 935.681 microseconds.
Using 2262 range queries.
Reporting 1130 valid air routes.

Time for Turkish Airlines (63M passengers) join query = 580.196 microseconds.
Using 1309 range queries.
Reporting 640 valid air routes.

Time for Lufthansa (62M passengers) join query = 827.419 microseconds.
Using 1846 range queries.
Reporting 919 valid air routes.
```

Total valid air routes count for the top ten passenger carriers = 15433.
Time for all ten standard join queries, in total = 13575.730 microseconds.
Using a total of 31096 range queries.

Second, we create a full memoized join of all the routes for all airlines in very compact form.

Time for full memoized join creation = 69234.922 microseconds.
Using 139697 range queries.
Full memoized join: valid routes count for all airlines = 65612

Third, we use that full memoized join to do ten memoized join queries listing the valid routes for the top ten passenger carriers in the world (as of 2016). Note that the numbers of routes are the same as the nested loop joins, but take three orders of magnitude less time, and only one range query for each.

Time for American Airlines (199M passengers) join query = 7.957 microseconds.
Using 1 range query.
Reporting 2340 valid air routes.

Time for Delta Air Lines (184M passengers) join query = 4.126 microseconds.
Using 1 range query.
Reporting 1977 valid air routes.

Time for Southwest Airlines (152M passengers) join query = 4.374 microseconds.
Using 1 range query.
Reporting 1140 valid air routes.

Time for United Airlines (143M passengers) join query = 4.311 microseconds.
Using 1 range query.
Reporting 2172 valid air routes.

Time for Ryanair/Ireland (120M passengers) join query = 8.330 microseconds.
Using 1 range query.
Reporting 2482 valid air routes.

Time for China Southern Airlines (85M passengers) join query = 3.824 microseconds.
Using 1 range query.
Reporting 1422 valid air routes.

Time for China Eastern Airlines (81M passengers) join query = 3.264 microseconds.
Using 1 range query.
Reporting 1211 valid air routes.

Time for easyJet/UK (73M passengers) join query = 3.283 microseconds.
Using 1 range query.
Reporting 1130 valid air routes.

Time for Turkish Airlines (63M passengers) join query = 2.519 microseconds.
Using 1 range query.
Reporting 640 valid air routes.

Time for Lufthansa (62M passengers) join query = 4.960 microseconds.
Using 1 range query.
Reporting 919 valid air routes.

Total valid air routes count for the top ten passenger carriers = 15433.
Time for all ten memoized join queries, in total = 46.948 microseconds.
Using a total of 10 range queries.

Statistics for the run of the program, including:

1. Allocating and loading three tables from CSV files on disk, around 80K rows.
2. Allocating, copying the keys and sorting ten indexes on those three tables.
3. Creating three relation vectors, each containing a from-index and a to-index.
4. Creating a tuple from those three relation vectors to store the query objects.
5. Creating and optimizing a database join query plan for those three relation vectors.
6. Creating a joined row output array and doing ten nested loop join queries into it for the top ten airlines.
7. Doing a full memoized loop join query for all airlines into it.
8. Doing ten memoized join queries into arrays for the top ten airlines.

```
Total run time spent in main() procedure thus far = 0.20185 seconds

Total relational database system time spent in main() procedure thus far, including I/O = 0.20105 seconds

Total display output time spent in main() procedure thus far = 0.00080 seconds
_____

Segmentation fault (core dumped)
[charles@localhost Kaldanes]$
```

BigJoinDemo output

The **BigJoinDemo** program builds the same database as **JoinDemo** with five **RelationVector** types, repeating one making six in the join order, and with six formal parameter objects in that same join order then thrice performs an identical set of ten queries for the an even distribution of the double routes by passenger carriers from the largest to the smallest in the airline industry, first as nested loop joins, second as memoized joins and finally as memoized joins without moving the output arrays from the memoized join array of double routes in 2 runs: one for the worst case and one for the best case (the first run has cache polluted by the previous data move of 15MB.)

That memoized joined row data move is unnecessary, since the range query on the memoized joined row array has the same objects in the same order as moving it to an output array: without the data move makes an indistinguishable result, but much faster. For example, American Airlines has the largest set of 562,148 valid double routes of 28 bytes each, requiring a total block move of a little over 15MB of output array. The output objects are sextuplets (first airline, first route, first source airport, first destination airport = second source airport for the second route, second route, second destination airport, second airline) and are counted by the program, and not printed, because there are way too many.

At the scale of American Airlines, (1) the double route nested loop joins take 0.415 second, (2) the double route memoized joins execute in 4.32 milliseconds since the output arrays created are large (as noted above) and are nearly two orders of magnitude faster than the double route nested loop joins (milliseconds versus tenths of seconds), and the double route memoized joins without the unnecessary move take 1.94 microseconds in the second run. That comes to over five orders of magnitude faster than the double route nested loop joins and over three orders of magnitude faster than the double route memoized joins with the block move.

The memoized join without the data move is arbitrarily scalable, since increasing the size of the memoized joined row array by 3 orders of magnitude only increases the join time by a half microsecond (best case) or one microsecond (worst case.) And since increasing the width of the joined row element (more tables in the join) does not increase the number of hops in the binary search, adding more tables to the join does not increase join query access time at all.

However that is, after the output result arrays are created, the base processing cost (loading and storing to move it) for any single joined row resulting array element by indexing that array slab is estimated at 2-4 nanoseconds.

```
[charles@localhost Kaldanes]$ ./BigJoinDemo


[Note: Remember to set the C++11 switch in the IDE or compiler!

 Remember to use release builds if you are analyzing performance,
   otherwise Index builds will be very slow!

 Remember to set the "ulimit -s" soft and hard stack limits to unlimited,
   otherwise it can die!]


Time for space allocating and loading the three tables from CSV files = 29990.409 microseconds.
Time for space allocating the ten indexes = 0.728 microseconds.
Time for copying the keys and sorting the ten indexes = 87376.175 microseconds.
Time for each index on the average = 8737.618 microseconds.
Time for creating the six relation vectors = 0.467 microseconds.
Time for creating the tuple containing the six relation vectors = 0.459 microseconds.
Time for creating the query plan to join the six relation vectors = 5.024 microseconds.
Time for creating the joined row array to contain the join output = 0.625 microseconds.
_____________________________________________________________________________

First, we do ten standard seven-table join queries listing the valid double routes (routes possessing valid airlines, from and to airports ... linking to a second route
from the first destination airport on to any second airport on any airline) for the ten lowest-to-highest route carriers to show the effects of scale.

Time for  Alaska Central Express join query = 109.252 microseconds.
Using 128 range queries.
Reporting 60 valid double air routes.

Time for Yangon Airways join query = 1455.412 microseconds.
Using 2660 range queries.
Reporting 1310 valid double air routes.

Time for Air Armenia join query = 1687.652 microseconds.
Using 3090 range queries.
Reporting 1512 valid double air routes.

Time for Airnorth join query = 678.123 microseconds.
Using 1467 range queries.
Reporting 682 valid double air routes.

Time for Ciel Canadien join query = 4296.249 microseconds.
Using 8840 range queries.
Reporting 4267 valid double air routes.

Time for Thomas Cook Airlines join query = 17421.524 microseconds.
Using 37178 range queries.
Reporting 18210 valid double air routes.

Time for Lion Mentari Airlines join query = 11036.945 microseconds.
Using 26346 range queries.
Reporting 12694 valid double air routes.

Time for Scandinavian Airlines System join query = 73939.499 microseconds.
Using 181933 range queries.
Reporting 89503 valid double air routes.

Time for Air France join query = 244547.142 microseconds.
Using 668397 range queries.
Reporting 331632 valid double air routes.

Time for American Airlines join query = 415044.200 microseconds.
Using 1135710 range queries.
Reporting 562148 valid double air routes.
```

Total valid double air routes count for the ten lowest-to-highest route carriers = 1022018.
Time for all ten standard join queries, in total = 770215.998 microseconds.
Using a total of 2065749 range queries.

Second, we create a full memoized seven-table join of all the valid double routes for each airline in very compact form.

Time for full memoized join creation = 11011065.170 microseconds.
Using 22048008 range queries.
Full memoized join: valid routes count for all airlines = 10817108

Third, we use that full memoized join to do ten memoized seven-table join queries listing the valid double routes for the ten lowest-to-highest route carriers to show the effects of scale, including the cost of moving the bytes although we don't need to (it's only for an apples to apples comparison.) Note that the numbers of double routes are the same as the nested loop joins, but take orders of magnitude less time, and only one range query for each.

Time for Alaska Central Express join query = 8.446 microseconds.
Using 1 range query.
Reporting 60 valid double air routes.

Time for Yangon Airways join query = 19.081 microseconds.
Using 1 range query.
Reporting 1310 valid double air routes.

Time for Air Armenia join query = 15.925 microseconds.
Using 1 range query.
Reporting 1512 valid double air routes.

Time for Airnorth join query = 9.417 microseconds.
Using 1 range query.
Reporting 682 valid double air routes.

Time for Ciel Canadien join query = 37.729 microseconds.
Using 1 range query.
Reporting 4267 valid double air routes.

Time for Thomas Cook Airlines join query = 134.998 microseconds.
Using 1 range query.
Reporting 18210 valid double air routes.

Time for Lion Mentari Airlines join query = 83.494 microseconds.
Using 1 range query.
Reporting 12694 valid double air routes.

Time for Scandinavian Airlines System join query = 660.048 microseconds.
Using 1 range query.
Reporting 89503 valid double air routes.

Time for Air France join query = 2561.105 microseconds.
Using 1 range query.
Reporting 331632 valid double air routes.

Time for American Airlines join query = 4320.658 microseconds.
Using 1 range query.
Reporting 562148 valid double air routes.

Total valid double air routes count for the ten lowest-to-highest route carriers = 1022018.
Time for all ten memoized join queries, in total = 7850.901 microseconds.
Using a total of 10 range queries.

Fourth, let's execute a range query boundary lambda from the airlineId index, for all double routes from each airline using the memoized join, to find out the average cost of getting the low and high range query boundaries.

Time for determining all airlines range query boundaries (not touching the rows) on the routes table = 4808.405 microseconds.
Using a total of 6162 range queries (number of airlines.)
Yielding an average time of 0.780 microseconds to find one range query boundary pair.

Finally, we use that full memoized join to do ten memoized seven-table join queries listing the valid double routes for the ten lowest-to-highest route carriers to show the effects of scale: however, without the unnecessary byte moves (this is an apples to oranges comparison, but you can access the bytes from the memoized join and don't need to output them.) Note that the numbers of these double routes are the same as the nested loop joins, but take effectively constant time in comparison, and only one range query for each. We do two runs for comparison, because the measurement is so small that the second run is different in an interesting way.

First run.

Time for Alaska Central Express join query = 2.736 microseconds.
Using 1 range query.
Reporting 60 valid double air routes.

Time for Yangon Airways join query = 2.032 microseconds.
Using 1 range query.
Reporting 1310 valid double air routes.

Time for Air Armenia join query = 2.454 microseconds.
Using 1 range query.
Reporting 1512 valid double air routes.

Time for Airnorth join query = 2.264 microseconds.
Using 1 range query.
Reporting 682 valid double air routes.

Time for Ciel Canadien join query = 1.682 microseconds.
Using 1 range query.
Reporting 4267 valid double air routes.

Time for Thomas Cook Airlines join query = 2.424 microseconds.
Using 1 range query.
Reporting 18210 valid double air routes.

Time for Lion Mentari Airlines join query = 2.425 microseconds.
Using 1 range query.
Reporting 12694 valid double air routes.

Time for Scandinavian Airlines System join query = 2.102 microseconds.
Using 1 range query.
Reporting 89503 valid double air routes.

Time for Air France join query = 2.479 microseconds.
Using 1 range query.
Reporting 331632 valid double air routes.

Time for American Airlines join query = 2.620 microseconds.
Using 1 range query.
Reporting 562148 valid double air routes.

Total valid double air routes count for the ten lowest-to-highest route carriers = 1022018.
Time for all ten memoized join queries, in total = 23.218 microseconds.
Using a total of 10 range queries.

Second run.

Time for Alaska Central Express join query = 1.785 microseconds.
Using 1 range query.
Reporting 60 valid double air routes.

Time for Yangon Airways join query = 2.005 microseconds.
Using 1 range query.
Reporting 1310 valid double air routes.

Time for Air Armenia join query = 2.000 microseconds.
Using 1 range query.
Reporting 1512 valid double air routes.

Time for Airnorth join query = 1.901 microseconds.

```
Using 1 range query.
Reporting 682 valid double air routes.

Time for Ciel Canadien join query = 1.707 microseconds.
Using 1 range query.
Reporting 4267 valid double air routes.

Time for Thomas Cook Airlines join query = 1.975 microseconds.
Using 1 range query.
Reporting 18210 valid double air routes.

Time for Lion Mentari Airlines join query = 1.830 microseconds.
Using 1 range query.
Reporting 12694 valid double air routes.

Time for Scandinavian Airlines System join query = 1.819 microseconds.
Using 1 range query.
Reporting 89503 valid double air routes.

Time for Air France join query = 1.997 microseconds.
Using 1 range query.
Reporting 331632 valid double air routes.

Time for American Airlines join query = 1.944 microseconds.
Using 1 range query.
Reporting 562148 valid double air routes.

Total valid double air routes count for the ten lowest-to-highest route carriers = 1022018.
Time for all ten memoized join queries, in total = 18.963 microseconds.
Using a total of 10 range queries.

_____

Statistics for the run of the program, including:
  1. Allocating and loading three tables from CSV files on disk, around 80K rows.
  2. Allocating, copying the keys and sorting ten indexes on those three tables.
  3. Creating six relation vectors, each containing a from-index and a to-index.
  4. Creating a tuple from those six relation vectors to store the query objects.
  5. Creating and optimizing a database seven-table double-route join query plan for those six relation vectors.
  6. Creating a joined row output array and doing ten nested loop join queries into it for the double-routes of the ten lowest-to-highest route carriers.
  7. Doing a full memoized loop join double-route query for all airlines into the joined row output array.
  8. Doing ten memoized join double-route queries into arrays for the ten lowest-to-highest route carriers (apples to apples comparison.)
  9. Doing ten memoized join double-route queries for the ten lowest-to-highest route carriers (apples to oranges comparison.)

Total run time spent in main() procedure thus far = 11.91265 seconds

Total relational database system time spent in main() procedure thus far, including I/O = 11.91136 seconds

Total display output time spent in main() procedure thus far = 0.00129 seconds
_____

[charles@localhost Kaldanes]$
```